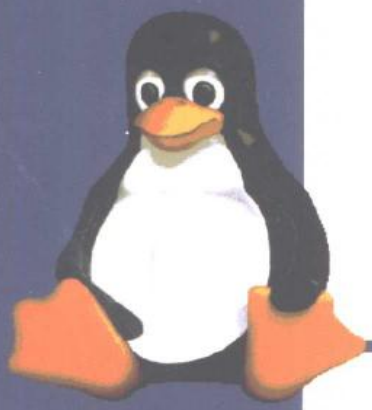
Linux专家之路



# Linux下的Perl编程

郑武 老马 编著　　雨人科技 策划

人民邮电出版社
www.pptph.com.cn

附光盘
CD-ROM

Linux专家之路



# Linux下的Perl编程

郑武 老马 编著 雨人科技 策划

人民邮电出版社

**图书在版编目（CIP）数据**

Linux 下的 Perl 编程 / 郑武，老马编著. —北京：人民邮电出版社，2001.11
（Linux 专家之路）
ISBN 7-115-09787-9

I. L… II. ①郑… ②老… III. PERL 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2001）第 074383 号

## 内 容 提 要

本书是一本介绍 Linux 平台上 Perl 5 编程语言的专业参考书。书中详细地讲解了 Perl 语言的基本概念、基本数据结构和语法规则，探讨了使用 Perl 语言进行 CGI 编程技术，还讲述了如何使用 Perl 语言进行数据库的开发，最后介绍了 Perl 语言在网络中的应用，讲解如何使用 Perl 语言进行网络编程。

本书可用作 Perl 5 初级用户的入门教程，也可作为中高级用户的 Perl 5 快速查询手册。

TG5180/61

Linux 专家之路

**Linux 下的 Perl 编程**

◆ 编　　著　郑　武　老　马
策　　划　雨人科技
责任编辑　张瑞喜
执行编辑　郭立罡

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ pptph.com.cn
网址　http://www.pptph.com.cn
读者热线:010-67129212　010-67129211(传真)
北京汉魂图文设计有限公司制作
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本:787 × 1092　1/16
印张:28.75
字数:696 千字　　2001 年 11 月第 1 版
印数:1 - 4 000 册　　2001 年 11 月北京第 1 次印刷
ISBN 7-115-09787-9/TP·2536

定价:48.00 元(附光盘)

本书如有印装质量问题,请与本社联系 电话:(010)67129223

# 前　言

随着因特网的普及，掌握一门网络编程语言已经成为用户一种迫切的需要，而 Perl 5 正是当前网络编程中最常用的语言。与其他语言相比，Perl 5 具有明显的优势：语法简单灵活，可移植性好，具有强大的字符串处理能力，可以进行高效率的 CGI 编程。尤其值得一提的是，由于 Perl 的开放性，读者可以方便地从网上免费获得大量可重用的模块和源代码等资源。

本书深入浅出地讲解了 Perl 5 的基础知识及其高级应用。全书共分 17 章，第 1 章介绍了 Linux 操作环境和 Perl 5 编程语言的基本情况；第 2 章到 14 章讲解了 Perl 5 的基础知识，首先介绍了 Perl 5 的 3 种基本数据结构、操作符和控制语句，接着详细介绍了 Perl 5 最强大的功能——正则表达式和字符串处理，另外还介绍了函数、引用以及 I/O 操作和文件系统，在介绍了包和模块后，还对 Perl 5 中面向对象的编程进行了讨论，每个章节都有丰富的例子，读者通过它们可以很快地掌握 Perl 5 的使用。第 15 章到第 17 章分别介绍了 Perl 5 中的 CGI 编程、DBI 数据库编程和网络编程等 Perl 5 的高级应用。

本书每章后面均附有习题，以便读者练习掌握所学知识，并在书后面的附录中列有习题的参考答案。

本书配有光盘。光盘内容包括程序源代码，部分习题参考答案，雨人科技技术支持文档，PERL COMPILER、PERL EDITOR 和 WEB SERVER。

郑武负责全书的设计、统稿和修改，并编写了第 1 章、第 13 章、第 15 章、第 16 章和第 17 章，还负责了光盘内容的制作和例子的测试；老马负责编写了第 2 章至第 12 章和第 14 章；参加本书编排和指导工作的还有雨人科技的刘馨宇等同志。在此对所有关心、支持和帮助过本书编写工作的同志表示诚挚的谢意！

书中不足之处敬请读者指正，以便再版时修订。

联系邮件地址：yurennet@263.net

雨人科技网站：www.yurennet.com

编著者

2001 年 8 月

# 目　录


**第 1 章　Linux 及 Perl 概述** …… 1

1.1　Linux 系统简介 …… 2
1.1.1　什么是 Linux …… 2
1.1.2　Linux 的发展历史 …… 2
1.1.3　Linux 主要发行版本 …… 3
1.1.4　Linux 的应用 …… 6
1.1.5　谁在使用 Linux …… 7
1.2　Perl 介绍 …… 8
1.2.1　Perl 概述 …… 8
1.2.2　Perl 发展历史 …… 10
1.2.3　Perl 的安装 …… 11
1.2.4　Perl 资源 …… 12
1.3　小结与练习 …… 13
1.3.1　小结 …… 13
1.3.2　习题与思考 …… 13

**第 2 章　标量数据** …… 15

2.1　标量变量简介 …… 16
2.1.1　何谓标量变量 …… 16
2.1.2　标量变量的命名 …… 17
2.2　数字标量 …… 17
2.2.1　整型表示法 …… 18
2.2.2　浮点型表示法 …… 20
2.2.3　数字格式 …… 21
2.2.4　定点数方案 …… 21
2.3　字符串标量 …… 22
2.3.1　单引号字符串 …… 22
2.3.2　双引号字符串 …… 23
2.3.3　反引号 …… 26
2.3.4　引用操作符 …… 27
2.4　常用的与标量有关的系统函数 …… 27
2.4.1　数学函数 …… 27
2.4.2　标量转换函数 …… 28

2.5 小结与练习 …… 29
2.5.1 小结 …… 29
2.5.2 习题与思考 …… 30

**第 3 章 数组和列表数据** …… 31

3.1 数组和列表的基本概念 …… 32
3.1.1 什么是数组和列表 …… 32
3.1.2 数组的表示 …… 32
3.1.3 列表变量 …… 34
3.2 数组元素的存取 …… 35
3.2.1 列表的赋值 …… 35
3.2.2 存取数组元素 …… 36
3.3 对数组元素的其他操作 …… 39
3.3.1 字符串中的方括号和变量替换 …… 39
3.3.2 数组的输出 …… 39
3.3.3 改变数组元素的值 …… 40
3.4 与数组有关的函数 …… 42
3.4.1 按字符顺序排序——sort 函数 …… 43
3.4.2 反转数组——reverse 函数 …… 44
3.4.3 数组去尾——chop 和 chomp 函数 …… 44
3.4.4 数组连接/拆分——join/split 函数 …… 44
3.4.5 push 和 pop 函数 …… 45
3.4.6 shift 和 unshift 函数 …… 46
3.4.7 拼接函数——splice 函数 …… 46
3.5 小结与练习 …… 47
3.5.1 小结 …… 47
3.5.2 习题与思考 …… 47

**第 4 章 散列** …… 49

4.1 散列简介 …… 50
4.1.1 什么是散列 …… 50
4.1.2 为什么要使用散列 …… 50
4.1.3 散列变量 …… 52
4.1.4 全局散列变量 …… 54
4.2 存取散列的元素 …… 54
4.2.1 访问散列中的元素 …… 54
4.2.2 增加元素 …… 55
4.2.3 数组与散列的相互转换 …… 56

4.3 散列函数 …… 57
4.3.1 keys 函数 …… 58
4.3.2 values 函数 …… 59
4.3.3 each 函数 …… 60
4.3.4 delete 函数 …… 60
4.4 散列应用举例 …… 62
4.4.1 用散列实现多维数组 …… 62
4.4.2 用散列实现结构 …… 64
4.5 小结与练习 …… 65
4.5.1 小结 …… 65
4.5.2 习题与思考 …… 65

**第 5 章 操作符** …… 67

5.1 数字操作符 …… 68
5.1.1 基本数字操作符 …… 68
5.1.2 数字比较操作符 …… 69
5.1.3 赋值操作符 …… 70
5.1.4 自增自减操作符 …… 72
5.2 字符串操作符 …… 74
5.2.1 字符串比较操作符 …… 74
5.2.2 字符串联接和重复操作符 …… 75
5.3 逻辑操作符和位操作符 …… 77
5.3.1 逻辑操作符 …… 77
5.3.2 位操作符 …… 78
5.4 其他操作符 …… 79
5.4.1 逗号操作符 …… 79
5.4.2 条件操作符 …… 80
5.5 操作符的优先级和结合性 …… 82
5.6 小结与练习 …… 84
5.6.1 小结 …… 84
5.6.2 习题与思考 …… 84

**第 6 章 控制结构** …… 87

6.1 控制结构中的有关问题 …… 88
6.1.1 语句块 …… 88
6.1.2 养成良好的编程习惯 …… 88
6.2 条件语句 …… 90
6.2.1 if 系列语句 …… 90

6.2.2 unless 语句 …… 94
6.2.3 if 语句的嵌套 …… 95
6.3 循环语句 …… 96
6.3.1 while/until 循环 …… 97
6.3.2 do while/until 循环 …… 99
6.3.3 for 循环 …… 100
6.3.4 foreach 循环 …… 101
6.4 其他控制语句 …… 103
6.4.1 标记修饰符 …… 103
6.4.2 修改循环控制流的几个语句 …… 104
6.4.3 表达式修饰符 …… 107
6.4.4 &&和|| …… 108
6.5 小结与练习 …… 109
6.5.1 小结 …… 109
6.5.2 习题与思考 …… 110

**第 7 章 正则表达式** …… 113

7.1 正则表达式简介 …… 114
7.1.1 什么是正则表达式 …… 114
7.1.2 一个简单的例子 …… 114
7.2 正则表达式的构成 …… 115
7.2.1 元字符 …… 115
7.2.2 原子 …… 116
7.3 正则表达式的正则 …… 116
7.3.1 单字符类型 …… 117
7.3.2 组合类型 …… 118
7.3.3 定位类型 …… 121
7.3.4 优先级 …… 122
7.3.5 正则表达式正则的应用举例 …… 122
7.4 模式修饰符 …… 127
7.4.1 全局匹配 …… 127
7.4.2 忽略大小写的匹配 …… 129
7.4.3 多行匹配与单行匹配 …… 130
7.4.4 单次匹配 …… 131
7.4.5 多余空格 …… 132
7.5 与正则表达式相关的其他问题 …… 133
7.5.1 模式匹配变量 …… 133
7.5.2 替代函数 …… 134

7.5.3 内插变量 …… 135
7.5.4 模式次序变量 …… 136
7.5.5 扩展模式匹配 …… 137
7.6 小结与练习 …… 138
7.6.1 小结 …… 138
7.6.2 习题与思考 …… 139

**第 8 章 字符串操作** …… 141

8.1 字符串和数值的相互转换 …… 142
8.1.1 数字与字符的相互转换 …… 142
8.1.2 数字与字符串的相互转换 …… 143
8.2 把标量插入字符串 …… 144
8.3 字符串处理函数 …… 145
8.3.1 改变大小写函数 …… 146
8.3.2 字符串截尾函数 …… 147
8.3.3 搜寻子字符串函数 …… 149
8.3.4 子字符串的展开与替代 …… 150
8.3.5 字符串转换函数 …… 153
8.3.6 取字符串长度函数 …… 154
8.3.7 分离和连接字符串函数 …… 155
8.4 小结与练习 …… 157
8.4.1 小结 …… 157
8.4.2 习题与思考 …… 158

**第 9 章 自定义函数** …… 161

9.1 一些常用的系统函数及其使用 …… 162
9.2 自定义函数的定义 …… 165
9.3 函数的调用 …… 166
9.3.1 先定义后调用 …… 167
9.3.2 用&调用 …… 167
9.3.3 前向引用 …… 168
9.3.4 用 do 调用 …… 168
9.3.5 函数调用的返回值 …… 168
9.4 更方便地使用函数 …… 170
9.4.1 在函数中使用参数 …… 170
9.4.2 在函数定义中使用私有变量 …… 171
9.4.3 使用半私有变量 …… 173
9.4.4 文件层级的 my 变量 …… 175

9.5 Perl 5 中预定义的函数 …… 176
9.6 函数的应用举例 …… 176
9.6.1 二进制和十进制之间的转换函数 …… 177
9.6.2 将角度转换为弧度 …… 177
9.6.3 求一个数的对数 …… 178
9.6.4 数字和英文的对应 …… 178
9.6.5 一个较复杂的例子 …… 180
9.7 小结与练习 …… 182
9.7.1 小结 …… 182
9.7.2 习题与思考 …… 182

**第 10 章 引用** …… 185

10.1 引用简介 …… 186
10.1.1 什么是引用 …… 186
10.1.2 引用操作符 …… 186
10.2 引用变量类型简介 …… 187
10.2.1 直接引用变量 …… 188
10.2.2 常量引用变量 …… 188
10.2.3 函数引用变量 …… 188
10.2.4 符号引用变量 …… 189
10.2.5 匿名引用变量 …… 190
10.2.6 对象引用变量 …… 192
10.3 直接引用变量的使用 …… 192
10.3.1 使用标量引用 …… 192
10.3.2 使用数组引用 …… 194
10.3.3 使用散列引用 …… 195
10.3.4 文件句柄的引用 …… 197
10.4 访问引用的其他方法 …… 198
10.4.1 使用大括号 …… 198
10.4.2 使用向右箭头操作符 …… 199
10.4.3 使用方括号 …… 199
10.4.4 多维数组 …… 199
10.4.5 利用引用传给函数多个数组以及返回多个数组或散列 …… 200
10.5 小结与练习 …… 202
10.5.1 小结 …… 202
10.5.2 习题与思考 …… 202

**第 11 章 文件系统** …… 205

11.1 文件操作 …… 206
11.1.1 打开/关闭文件 …… 206
11.1.2 读/写文件 …… 208
11.1.3 删除文件 …… 209
11.1.4 更改文件名 …… 209
11.1.5 产生/删除文件链接 …… 210
11.1.6 判断文件状态 …… 212
11.1.7 读取文件中的某一行 …… 215
11.2 目录操作 …… 216
11.2.1 产生/删除目录 …… 216
11.2.2 打开/关闭目录 …… 217
11.2.3 其他目录处理函数 …… 217
11.3 更改文件和目录的权限、所有权和时间戳 …… 219
11.3.1 更改权限 …… 219
11.3.2 更改所有权 …… 220
11.3.3 更改时间戳 …… 220
11.4 使用 DBM 文件 …… 222
11.4.1 打开/关闭 DBM 文件 …… 222
11.4.2 DBM 散列的使用 …… 223
11.5 小结与练习 …… 223
11.5.1 小结 …… 223
11.5.2 习题与思考 …… 224

**第 12 章 输入/输出处理** …… 225

12.1 基本输入 …… 226
12.1.1 来自 STDIN 的输入 …… 226
12.1.2 输入操作符 …… 227
12.1.3 @ARGV 数组 …… 227
12.2 基本输出 …… 229
12.2.1 print 函数 …… 229
12.2.2 格式化输出函数 sprintf 和 printf …… 229
12.3 格式化输出 …… 232
12.3.1 什么是格式 …… 232
12.3.2 怎么定义格式 …… 233
12.3.3 使用格式 …… 234
12.3.4 字段容器 …… 237
12.3.5 其他问题 …… 240
12.4 小结与练习 …… 241

12.4.1 小结 …… 241
12.4.2 习题与思考 …… 242

**第 13 章 包和模块** …… 243

13.1 使用 Perl 包 …… 244
13.1.1 包的定义 …… 244
13.1.2 包的引用和导入 …… 245
13.1.3 包之间的切换 …… 246
13.1.4 包和子程序 …… 247
13.1.5 用包定义私有数据 …… 247
13.1.6 访问符号表 …… 248
13.2 使用 Perl 模块 …… 250
13.2.1 创建模块 …… 250
13.2.2 导入模块 …… 251
13.2.3 模块使用举例 …… 252
13.3 Perl 标准模块 …… 257
13.4 小结与练习 …… 267
13.4.1 小结 …… 267
13.4.2 习题与思考 …… 267

**第 14 章 面向对象编程** …… 269

14.1 对象与类简介 …… 270
14.1.1 对象的特征 …… 270
14.1.2 对象类 …… 270
14.1.3 对象方法 …… 271
14.1.4 对象变量 …… 271
14.1.5 对象引用变量 …… 271
14.2 用 Perl 5 创建类 …… 272
14.2.1 构造函数 …… 273
14.2.2 方法 …… 277
14.2.3 Perl 类中的成员变量 …… 281
14.2.4 析构函数 …… 282
14.3 继承 …… 283
14.4 方法的重载 …… 287
14.4.1 子类方法的重载 …… 287
14.4.2 操作符的重载 …… 289
14.5 关于 Perl 面向对象编程的一些其他问题 …… 292
14.6 小结与练习 …… 295

14.6.1 小结 …… 295
14.6.2 习题与思考 …… 296

**第 15 章 CGI 编程** …… 297

15.1 Web 体系结构与 CGI 程序结构 …… 298
15.1.1 Web 体系结构 …… 298
15.1.2 CGI 程序结构 …… 299
15.2 HTTP 协议 …… 303
15.2.1 HTTP 通信模型 …… 303
15.2.2 HTTP 请求 …… 304
15.2.3 HTTP 响应 …… 307
15.3 CGI 规范 …… 308
15.3.1 CGI 程序工作原理 …… 308
15.3.2 CGI 方法 …… 309
15.3.3 环境变量 …… 311
15.4 CGI 模块 …… 314
15.4.1 使用 CGI 模块 …… 314
15.4.2 创建动态文档 …… 315
15.4.3 创建 HTML 元素 …… 317
15.4.4 创建表单 …… 321
15.5 计数器实现 …… 331
15.5.1 设计思路 …… 331
15.5.2 代码实现 …… 334
15.6 留言簿实现 …… 335
15.6.1 设计思路 …… 335
15.7 小结与练习 …… 343
15.7.1 小结 …… 343
15.7.2 习题与思考 …… 343

**第 16 章 数据库编程** …… 345

16.1 数据库基础 …… 346
16.2 SQL 语句 …… 347
16.2.1 关系型数据库简介 …… 347
16.2.2 数据类型 …… 348
16.2.3 创建和删除表 …… 349
16.2.4 插入数据 …… 350
16.2.5 数据查询 …… 351
16.2.6 修改数据 …… 355

16.3 使用 DBI …… 356
16.3.1 DBI 与 DBD …… 356
16.3.2 DBI 编程环境的搭建与测试 …… 357
16.3.3 句柄 …… 360
16.3.4 连接和断开连接 …… 361
16.3.5 数据库操作 …… 364
16.4 DBM 数据库 …… 376
16.4.1 创建和打开 DBM 文件 …… 377
16.4.2 插入和删除记录 …… 378
16.4.3 遍历 DBM 数据库 …… 379
16.5 小结与练习 …… 381
16.5.1 小结 …… 381
16.5.2 习题与思考 …… 381

**第 17 章 网络编程** …… 383

17.1 Socket 基础 …… 384
17.1.1 Socket 原理 …… 384
17.1.2 客户端 / 服务器模型 …… 385
17.2 Perl 对 Socket 的支持 …… 388
17.2.1 和协议相关的函数 …… 388
17.2.2 Socket 底层函数 …… 394
17.3 面向连接模型应用 …… 399
17.3.1 服务器端的程序 …… 400
17.3.2 客户端的程序 …… 403
17.4 使用预编译模块 …… 404
17.4.1 使用 Net::FTP 模块 …… 405
17.4.2 使用 Net::POP3 模块 …… 405
17.5 RPC 的实现 …… 406
17.5.1 Msg 模块实现 …… 406
17.5.2 Msg 模块使用举例 …… 417
17.5.3 RPC 模块的实现 …… 418
17.5.4 RPC 模块使用举例 …… 421
17.6 小结与练习 …… 422
17.6.1 小结 …… 422
17.6.2 习题与思考 …… 422

**附录 部分习题参考答案** …… 423

# 第1章

# Linux及Perl概述

Linux系统简介

Perl介绍

小结与练习



从 Linux 的诞生到现在，经历了这么多年时间，我们对 Linux 这个名词听起来已经不再陌生了，有许许多多的人在学习和使用 Linux 操作系统，或在 Linux 系统上进行应用程序的开发。

Linux 是一个相当优秀的操作系统资源，这主要是因为 Linux 操作系统本身是免费的，而且用户可以获得它的源代码。此外，Linux 下的许多开发工具也是免费的，这些工具的源代码都是公开的，这意味着用户可以通过研究现有的代码来提高自己的编程水平。众所周知，学习程序设计最好的方法就是看专家写的程序，然后在上面修修改改（当然绝大多数的修改是为了自己对程序理解的更加深刻，而不是修改程序中的“bug”）然后调试，这样可以很快地提高自身的程序设计水平。

Perl 是一门很好的编程语言，它的功能非常强大，而且在许多操作系统上都有它的编译程序，Linux 操作系统也不例外。本书所要研究的就是在 Linux 操作系统上 Perl 语言的编程。

本章要点：

- Linux 系统简介。
- Perl 介绍。

# 1.1 Linux 系统简介

## 1.1.1 什么是 Linux

Linux 是一个稳定的、功能强大的而且免费的操作系统。

Linux 是 Internet 的产物，最初它属于一个学生（Linus Torvalds）的个人爱好，而后来却演变成为一个最流行的免费操作系统。对许多人而言，Linux 似乎是个谜，一个免费的东西怎么会有价值？在一个被一群软件巨头统治的（系统）软件王国里，一个由一些电脑黑客编写的操作系统如何能够参与竞争？一个由不同的国家不同的人编写的软件如何能够保持其稳定性和高效性？其答案是肯定的。Linux 具有非常好的可靠性、高效性和竞争能力，许多大学和研究机构都在用 Linux 来作计算；许多人已在其个人计算机上安装了 Linux；绝大多数公司都在或多或少地使用 Linux。Linux 被广泛地用来浏览 Web 站点，文件处理，发送 E-mail，玩计算机游戏等等。Linux 绝不是一个计算机界的玩具，而是一个由全世界的爱好者共同开发的非常完善的、专业化的操作系统。

## 1.1.2 Linux 的发展历史

Linux 的源头可以追溯到 UNIX 家族。1969 年，贝尔实验室的研究人员 Ken Thompson 开始在一台空闲的 PDP-7 机器上实验其多用户、多任务的操作系统（multi-user，multi-tasking operating system），不久 Dennis Richie 和其他两位同事加入了他的行列，他们与实验室中的其他同事一道开发出了最早期的 UNIX 版本。早期的 UNIX 是用汇编语言编写的，第 3 版时采

用了 C 语言。C 语言是 Richie 设计并编写的，以用来作为编写操作系统的语言。用 C 语言改写过的 UNIX 便可以被移植到 PDP-11/45 和 DIGITAL 11/70 计算机上，这是 UNIX 操作系统的一个历史性的转折，使得 UNIX 正式从实验室走向大型机计算环境。很快，绝大多数的计算机制造商都发布了其相应的 UNIX 版本。

Linux 诞生的原因极其简单。Linus Torvalds 是 Linux 的作者和主要管理者，当时仅仅是一名学生，他当时使用的操作系统是 Minix。Minix 是一个非常简单、UNIX 风格、被广泛应用在教学上的操作系统。Linus 对 Minix 的功能不是很满意，决定自己动手编写一个软件。因为在学校里每天使用的都是 UNIX，所以他选择将 UNIX 作为软件的模型。最开始的工作是在一台 Intel 386 的 PC 机上，他的进展很迅速。Linus 对他所做的事情充满了兴趣，并通过刚刚出现的、还局限在学术领域的计算机网络，将已有的代码共享给其他的学生。许多人看见了 Linus 的软件很有兴趣并开始加入到开发的行列。不同的人由于在使用 Linux 时碰到不同的问题，所以这个软件也就不断地被更新和完善。不久之后，Linux 就成为一个完整的操作系统。值得注意的是，Linux 中没有任何 UNIX 代码，而是根据 POSIX 标准重新编写的，Linux 中使用了许多在 Cambridge、Massachusetts 的 Free Software Foundation 提供的 GNU 软件。

多数人仅把 Linux 当做一个简单的工具来使用，也有许多用户在 Linux 上进行应用程序的开发，只有很少数人敢于为 Linux 编写设备驱动程序和系统核心的补丁（patches）。Linus Torvalds 接受来自任何人和任何地方的关于系统核心的补充和修改，但是 Linus 严把质量关并由他自己将新的代码加入核心。当然，在任何时候，从事 Linux 核心开发的人员毕竟是只有一少部分人。

大多数 Linux 用户似乎不关心这个操作系统是如何构造和运行的。这不是个明智的决定，因为学习 Linux 是更好地理解操作系统功能的有利途径。Linux 不仅设计得很好，更重要的是其源代码是公开的。这是因为虽然作者拥有这个软件的版权，但在 Free Software Foundation's GNU Public License 的基础上，源代码是可以免费获取的。

对刚接触源代码的人，当他们看见一些叫做 kernel、mm 和 net 的子目录时，可能会觉得迷惑，它们含有些什么？这些代码是如何工作的？为了解决上述问题，需要做的是对整个 Linux 的结构有个总体的了解。

源代码公开有什么好处？举个例子，假如用户对 foxmail 之类的邮件收发程序感兴趣，可是又不知道它是怎么实现的，这个时候，一个很好的办法就是去找 Linux 下的自由软件，查看它的源代码，仔细研究它，最后肯定会知道这个软件是如何实现的。

### 1.1.3 Linux 主要发行版本

与 Microsoft Windows 98 或 NT 不同，Linux 操作系统并不是一个单一的产品，在市场上有多种发行版本，它们并不都是一样的。

所有的发行版本具有同样的 Linux 内核，内核包含了所有核心的操作系统功能以及网络堆栈。另外，他们都提供有标准的工具、一系列的应用程序、一些打印的文档以及有限的技术支持。每种发行版本都带有 X Window 系统以及一个图形用户界面、Web Server、e-mail

server 以及 FTP Server 等。真正不同的地方在于其安装、配置、支持以及第三方的应用。

例如，安装过程可能是繁琐的，在设置像网卡这样的硬件时用户可能会遇到棘手的困难。多数的 Linux 网络配置程序都有一个特定的命令行界面，并且需要用户具备专业知识，而技术支持是有限的。因此，尽管 Linux 发行版中带有许许多多的应用程序，但是没有一个像 Microsoft Office 这样成熟易用。不过，如果读者愿意投入时间去琢磨它，最终还是可以将一切配置好的。如果用户是刚刚接触 Linux，Red Hat 版本应该是最好的选择，因为它带有较为成熟的安装程序。Caldera 版本的安装程序无法与 Red Hat 相比，但是它带有一些极具吸引力的商业应用，非常适应中小型企业的需要。

下面介绍几款著名的 Linux 系统的发行版本。

1. Caldera OpenLinux

在人们的印象中，Linux 不是一个适合公司集团使用的操作系统，但是 Caldera 正试图改变这一状况。Caldera OpenLinux 是唯一带有全面商业应用的发行版，其中包含了它自己的 NetWare 客户端软件、一个数据库服务器（Sybase ASE）以及完整的商务软件。

与 Red Hat 类似，Caldera 也提供有启动盘，可以简化安装。总的来说，Caldera 的安装不如 Red Hat 那样直观，不过，除了需要使用 Fdisk 进行分区之外，Caldera 的安装似乎也不是很难，也不需要用户太多的干预。Caldera 包括了 Red Hat Package Manager（RPM），使得在 Linux 下安装和反安装应用程序变得容易（至少在 Linux 的标准上如此）。LISA（Linux Instalation System Administration）是一个图形化的工具，用于配置基本的网络客户端参数以及其他一些特性。当然，在实际应用中，还是需要花不少时间手动编辑配置文件。

Caldera 最强大的优势之一是 KDE（Kool Desktop Environment），这是一个图形化的桌面环境，使 Linux 在外观和操作上与 Windows 类似，而且还带有由 Star Division 开发的 StarOffice ，这是一个与 Microsoft Office 类似的套装软件，这样就有了可适用于中小企业的完整解决方案了。最后，Caldera 还提供多种形式的技术支持方式，包括电话和 e-mail 支持。

2. Debian GNU/Linux

Debian 是最难安装的发行版本。该系统没有启动盘，不过用户可以自己制作一张，如果系统允许，也可以从 CD 盘直接启动。安装程序带有 Fdisk 用以处理分区问题，这个安装程序部分上是图形化的，但是仍然不容易使用。如果用户对机器的每个硬件都很了解而且知道如何配置网络，这种类似于向导的安装过程还是可以接受的。在安装完成后，用户可能会发觉系统中几乎没有任何应用程序，emacs 文本编辑器以及 Web 浏览器必须用 Debian 的 dselect 程序安装，而 dselect 程序向来以难以使用而闻名。

当完成安装后，Debian 包含的 X Window 可以使工作稍微容易一些。该发行版最大的优点在于包括一个可选的包管理器——Debian Package Management System，它可以在安装之前对新的应用程序进行扫描，并且检查系统现有的配置情况，以决定所需安装的包，以免发生冲突。

3. Linux Slackware

这曾是一个最流行的 Linux 发布版本，由 Patrick Volkerding's software 制作，不过其市场领导地位现在已经让位于制作更精细的 Red Hat 了，可是它在老牌 Linux 用户中还是最流行的。与 Debian 一样，Linux Slackware 没有启动盘。当从映像（image）中制作启动盘时，众多的硬件选项令人生畏。

Slackware 不如 Caldera 或者 Red Hat 那样易于使用，但是还是有些优点的。如果在安装过程中忘记将某些设置写入 config script 就重启动了，Slackware 将自动向 root 用户发一个 e-mail，说明问题所在以及该如何恢复，它还提供了一批不错的 configuration scripts，可以自动完成一系列的命令。

Slackware 最好的特性是在其 CD 盘上有一套已经安装好的 Linux 和 X Window，如果系统支持从 CD 盘启动，可以直接试用 Linux 而无须往硬盘上安装任何东西。Slackware 带有一个转换工具可以管理 RPM 包以及一个窗口管理器 FVWM95。Slackware 原本由 Walnu Creek CDROM 发行。

4. Official Red Hat Linux

Official Red Hat Linux 已经成为 Linux 市场中最重要的一员，这主要是因为它提供了最优秀的安装程序以及先进的包管理程序。如果读者是第一次使用 Linux，Red Hat 是最好的选择。

像 Caldera 一样，Red Hat 包含了一张启动盘。安装程序将自动检测硬件，并且给出合乎逻辑的安装建议（笔记本电脑的用户一定会对其 PC Card 支持能力感到满意）。Red Hat 还提供了 WorkStation 和 Server 两种预设的安装模式，这些模式下，分区以及应用程序的选择都是设置好的，当然，也可以选择用户安装方式以自行决定分区方式。在分区方面，Red Hat 提供了一个 Disk Druid 应用程序，这个图形化的安装程序要优于 Fdisk，不过还是需要对分区和目录结构的基本概念有个了解。

为了使配置工作更容易，Red Hat 包含了一系列的工具，如用于设置 X Window 系统的 Xconfigurator 以及 XF86Setup，前者是一个向导型的配置程序；Linuxconf 可使配置网络变得更容易，包括配置 PPP 客户端、DHCP 以及 DNS；此外，如果你在安装过程中使用了 Red Hat 的 boot disk creation script，它将存储网络以及 XF86Config 信息，这样用户就不需要反复的输入这些信息了。

Red Hat 因其包管理程序（RPM）而闻名，这是一个开放源代码的程序，在其他许多发行版中也广泛应用。RPM 使安装和反安装应用变得安全，避免了程序冲突，甚至可以对内核本身进行升级而无须重装整个系统。

Linux 的版本有两种，一种指的是内核的版本，一种指的是发行的版本，所有的发行版本用的都是一个相同的内核（只是内核的版本不同而已）。内核版本号如果是偶数的，说明这个版本是已经经过了测试，是稳定的内核；如果内核版本号是奇数的，那么这个内核还处

于测试阶段，需要网络上的高手们不断对它进行修改、测试、发布补丁（patch）等。

### 1.1.4 Linux 的应用

Linux 的功能太强大了，很难说清楚它到底能做些什么，这里只是简单地介绍一下 Linux 最常见的应用。

1. Web 服务

Linux 被广泛应用于 Internet/Intranet 中，提供 WWW 服务，它的 WWW 服务器据说在全球的所有 WWW 服务器中占据了将近 60%到 70%，这多亏了 Apache 软件包，目前它的版本为 Apache 1.3.9。Wu-ftp 用来提供 FTP 服务、Sendmail 提供电子邮件服务、inn 提供新闻组服务等。另外，Linux 还被广泛用作 DNS 服务器、NIS 服务、NFS 服务等。

2. 文件及打印服务

长期以来，人们习惯了使用 NT 做文件服务器，但是，用过 Netware 的人都会发现，NT 所提供的文件服务功能太弱，比如，它无法对用户使用的磁盘进行定额限制，只要管理人员向用户开发了写入权限，一个贪婪的用户就可以将整个硬盘都写满数据。而使用 Linux 就不会这样，Linux 下的 samba 服务器，可以轻松面向用户提供文件及打印服务，还可以通过磁盘定额达到和 Netware 同样的目的。

3. 数据库服务

数据库服务是所有商业用户在选择系统时首先要考虑的问题，因为所有的商业系统的运作几乎没有人敢说可以不用数据库系统。Linux 提供了数据库系统的全力支持，早期的数据库系统有：MySQL、PostgreSQL、msql 等，其中 MySQL 可以支持到 5TB 以上的数据库。后来，随着 Linux 影响的不断扩大，各大数据库厂商纷纷看中 Linux 广阔的前景，先后推出了基于 Linux 系统下的大型数据库。现在，运行在其他环境下的大型数据库系统，几乎全部可以在 Linux 下运行。目前可以在 Linux 运行的大型数据库系统有 Sybase ASE 11.0.3.3、Sybase ASE 11.9.2、Informix-se、Informix Dynimic Server 7.3、Oracle 1.0.5 for Linux、Oracle 1.1.5 for Linux、IBM DB2 for Linux 6.1Beta、Notes/Domino 5.0.1 等。

4. 网络服务

如将 Linux 系统用作路由器时，这些系统通常情况下只需安装简单的具有网络服务功能的软件包即可。大多情况下是在一台“486”或较好一点的“586”计算机插上两块网卡，用作一个简易的路由器或网桥等，可节省很多不必要的开支。对于安全性要求较高的网络，还可以将 Linux 机当作防火墙（IP Chain）来使用，以杜绝外部入侵者的破坏。另外，Linux 还可以毫不费力地当作代理服务器（Socket、Squid 等），也可以采用 IP 掩蔽（IP Masquade）将整个局域网上用户全部接入 Internet。面向用户提供 WWW、FTP、EMAIL 等服务。

5. 典型应用

这是一个经济型的 ISP 解决方案。在所有的 PC 机上安装 Linux，每台 PC 机上安装 4 块硬盘，采用 md 将其以 raid0 方式拉成一个大盘，其他目录以 NFS 格式 export 输出；然后在其他 PC 机上同样安装 4 块硬盘，同样采用 raidtool 将其拉成一块大盘，以 NFS 格式 export 输出，最后，在主服务器上，通过在主服务器上安装（mount）NFS 文件系统，从而形成一个大的存储空间。WWW 服务器采用 Apache 1.3.12，数据库使用了 MySQL，通过 PHP 将 Web 同数据相连，使用户直接通过页面就可以访问数据库。以上种种，就构成了一个经济型的 ISP。

### 1.1.5　谁在使用 Linux

Linux 具有这么好的特性，而且是免费的，究竟都有些什么人在使用呢？可以从三个方面来分析。

1. 个人用户

这类用户大多为电脑爱好者或电脑发烧友，他们在媒体炒作的影响下，纷纷自发购置各种发行版，并急不可耐地将它们安装到自己的机器上。对于个人用户，安装使用 Linux ，或者是好奇，或者是抱着试试看的心态才安装 Linux 的，很少有用户在安装前就希望它能真正干点什么，这类用户中，学生占据了很大的比例。个人用户大多对 UNIX 不了解，更多的人在安装 Linux 系统之前，根本不曾接触过 UNIX，对于这类用户，看到长长的控制台命令就有点不知所措，遇到复杂的配置过程，大多数用户都是无计可施。个人用户在 Linux 的使用者中占据着极大的一部分，虽然，这部分用户大多只是想试用一下，或者说只是抱着“玩一玩”的态度，但随着 Linux 的进一步发展，这些用户将是 Linux 发展的潜在的最大用户群。随着 Linux 的各种发行版对系统配置的进一步简单化，个人用户的数量就会大大增加。

2. 专业用户

比较而言，专业用户就清醒得多。这类用户大多是 UNIX 的使用者，由于长期受商业 UNIX 的限制，他们一发现 Linux，就对它产生了好感。因为 Linux 的确可以干不少工作，而且它的伸缩性很强，用户可以随便更改它，所以他们特别喜欢 Linux。

对于专业用户，由于他们本身对 UNIX 比较熟悉，清楚地知道 Linux 究竟可以干什么，又由于有使用 UNIX 的经验，所以上手快，也可以发挥出 Linux 的强大功能。由于使用 Linux 不受软件版权的限制，专业用户可以获得所有的源程序代码，可以自由更改其中一些自己认为必要的部分，也可以对系统进行完全的量身定做，使系统达到高效运行状态。不过，目前国内这类用户的数量还比较少，更多的人则是从源程序中获得一些编程技巧，或是对内核提供的一些功能进行深入学习，从中找到一些有用的方法和思想等。专业用户是 Linux 最忠实的拥护者，不管媒体对 Linux 的看法是好是坏，专业用户永远不会受到丝毫的影响，因为他们从 Linux 的应用中得到真实的感受，那就是“它真的很好用，功能真正很强大！”。

3. 商业用户

商业用户由于要向客户提供商业服务，所以选择系统时特别认真，他们通常要考虑很多的因素，例如安全性、可靠性、费用等。综合各种因素，有的商业用户选择了 Linux ，这说明 Linux 已相当稳定，正在逐渐被商业用户所接受。当然，目前选择使用 Linux 系统的商业用户多为信息服务提供商，如大量的 ISP 或 ICP 等。由于人们对 Linux 的观望及不信任态度，目前 Linux 的商业用户还不够多，但是，随着 Linux 性能的提高，它将逐渐被广大的商业用户所认识。总有一天，它会同今天的 UNIX 一样，最终被人们所接受。

## 1.2 Perl 介绍

### 1.2.1 Perl 概述

Perl 是 Practical Extraction and Report Language 的缩写，它是由 Larry Wall 设计的，并由他不断更新和维护，用于在 UNIX 环境下编程。

Perl 具有高级语言（如 C 语言）的强大能力和灵活性。事实上，它的许多特性是从 C 语言中借用来的。与脚本语言一样，Perl 不需要编译器和链接器来运行代码，用户要做的只是写出程序并告诉 Perl 来运行而已。这意味着 Perl 对于小的编程问题的快速解决方案和为大型事件创建原型来测试潜在的解决方案是十分理想的。

Perl 提供有脚本语言（如 sed 和 awk）的所有功能，还具有它们所不具备的很多功能，Perl 支持从 sed 到 Perl 以及从 awd 到 Perl 的代码转换功能。

Perl 是按 GNU Public License 的形式分发的，其实质是免费的，开始仅运行于 UNIX 和类 UNIX 系统，现在可以方便地在 OS/2、Windows 9x、Windows/NT 等系统下运行。

Perl 是解释运行的。一般 Perl 程序的第一行需注明自己是一个 Perl 程序而不是 shell 程序，所以一般将下面一行语句作为文件的第一行。

```
#! /usr/bin/Perl
```

下面来看一看第一个 Perl 程序，源程序如下：

```
#! /usr/bin/Perl
print "Hello world! \n";
```

这里的第一行说明了这是一个 Perl 程序，它也是 Perl 的注释，注释是从#开始 至该行结束的所有文字；第二行是程序的可执行部分，这里只有一条 print 语句。

在此基础上做稍微复杂一点的改变，使该程序按名字打招呼。要做到这一点，需要一个

变量来存放姓名，使用$name 来保存姓名。

**程序清单 1-1：hello1.pl**

```
#!/usr/bin/Perl
printf "What 's your name?";
$name=<STDIN>;
chop($name);
printf "hello, $name，welcome!n";
```

这里第三行表示从终端获得行输入并赋值给$name，这里$name 的值有一个换行符，第四行“chop($name)”表示去掉$name 的最后一个字符（即换行符），第五行显示输入。

现在为 Perlstar 编写一个特殊的欢迎辞，而对其他人则采用普通欢迎辞，要达到这样的效果必须将输入的姓名与 Perlstar 作一个比较，如果相同则执行特殊功能。

**程序清单 1-2：hello2.pl**

```
#!/usr/bin/Perl
printf "what 's your name?";
$name=;
chop($name);
if ($name eq "Perlstar"){
   printf "hello,Perlstar welcome! \n";
}
else {
   printf "hello, $name，welcome! \n";
}
```

这里第五行用“eq”操作符对两个字符串进行比较，相同则执行下一句，否则执行 else 下的语句。

现在已经可以对 Perlstar 发出不同的欢迎辞了。但如何加上密码来控制权限呢？比如说，可以让除了 Perlstar 以外的人都必须输入密码直到正确为止。

**程序清单 1-3：hello3.pl**

```
#!/usr/bin/Perl
$password="wsxy";     #密码为wsxy
printf "what 's your name?";
$name=;
chop($name);
if ($name eq "Perlstar"){
```

```
    printf "hello ,Perlstar,welcome! \n";
}
else{
    printf "hello, $name!\n please enter password: ";
    $guess=;
    chop($guess);
        while ($guess ne $password){
         printf "password error , enter again: ";
     $guess=;
         hop($guess);
        }
}
```

这里先将密码保存在标量变量$password 中，然后当不是 Perlstar 的其他人登录后，先显示欢迎语句，然后要求输入密码，通过“ne”操作符将输入结果和$password 相比较，密码相同则退出，否则一直循环下去。

### 1.2.2 Perl 发展历史

为了理解 Perl 的发展，必须了解 Perl 为什么存在。

1986 年，Larry 还只是一个项目的系统管理员。这个项目的目标是能够支持多种安全级别的广域网络。他负责管理在西海岸的由 3 台 Vaxen 和 3 台 Sun 组成的实验设施，这些机器通过一条加密的 1200 波特率的串行线路连接到东海岸的类似设施。由于 Larry 的主要任务是提供技术支持，他能够利用他的优势去开发或增强各种各样的工具，比如 rn、patch 和 warp。有一天，老板对他说，“Larry，我们需要一个能控制和管理所有 12 台机器的系统，而且必须在一个月内完成。立即着手去做！”于是，从来不逃避工作的 Larry 开始考虑如何设计一个海岸间的控制管理系统。Larry 不是一个边做边想的人，于是他使用 B-news 系统来察看、维护和管理错误报告。

Larry 开始在这些机器上安装新闻系统，并增加了两条指令。其中“append”指令用于附加新的信息到已有的文章；而“synchronize”指令用于保证两个海岸的文章数目是相同的。管理和控制可以用 RCS 完成，余下的由 news 和 rn 完成。由于存储在不同文件中的新闻都要在主服务器上维护，因此文件有很多交叉引用。Larry 的第一个想法是“让我们用 awk”，但是，不幸的是那个年代的 awk 不能处理多个文件，Larry 不愿意为此编写一个用途单一的工具，于是一种新的语言诞生了。

这种新的工具最初并不叫 Perl。Larry 曾经将一些同事和合作者的名字搅在一起，也考虑过字典中所有的三四个字母的单词，不过他都不满意。经过多次改动，名字变成了 Pearl，最后终于变成了现在的 Perl，就这样，Perl 被延用至今（仍然可以从 Perl 的缩写解释中看到前一种称呼的遗迹，即“Practical Extraction and Report Language”）。

最初的 Perl 缺少很多现有的特性。虽然可以找到模式匹配、文件句柄、标量和文本格式。

不过当时只有很少的函数，甚至没有关联数组，只有从 rn 中借用来的有限的正则表达式，它的文档只有 15 页。不过 Perl 的执行比 sed 和 awk 都要快，因此在别的应用程序和项目中得到了应用。

后来，Larry 被调到了别的项目小组，很快发现它变成了系统管理的利器。并借用了 Henry Spencer 优秀的正则表达式包，把它加入了 Perl。随后 Larry 加入了各种自己需要的好东西，也加入了一小部分别人需要的东西。随后他在网络上发行了 Perl。

最令人兴奋的是，Larry 不断地发行新版本的 Perl，无论他是在喷气推力实验室、网络实验室还是在 O'Reilly。

### 1.2.3 Perl 的安装

要在 Linux 上安装 Perl ，需要具备两个条件。首先，需要一个 Perl 的源模块包。用户可以从 http://www.Perl.com 的 Downloads 区域下载最新版本。另外需要一个 GCC 编译器。如果不知道这个编译器的作用，也不必担心，Perl 的配置程序能够自动选定一个，如果没有这个编译器，可以安装一个预安装版本。

如果 Linux 配有一个用于安装预安装软件包的系统，用户就能够安装一个预安装的 Perl 版本。Linux、Solaris、AIX 和其他 UNIX 系统均配有已经捆绑的预安装 Perl 版本，它们的安装非常容易。当你拥有 Perl 的源模块包后（它的文件名类似 Stable.tar.gz），必须对它进行解包，然后进行安装。

具体的操作如下所示：

```
$ gunzip stable.tar.gz
$ tar xf stable.tar
```

这两个命令的运行需要花费一定的时间。如果没有 gunzip 解压缩程序，可以从 http://www.gnu.org 下载，该程序包称为 gzip。当完成所有的解包操作后，就会看到一个提示符，然后键入下面的命令：

```
$ sh Configure
```

这时，Configure 程序就开始运行了。此时，系统会提问一系列的问题，如果用户对大部分问题不知道如何回答也没有关系，只需要按回车键即可，系统默认的答案通常是最好的答案。Perl 几乎能够在任何 Linux 系统上安装而不会出现任何问题。当所有这些操作完成时，键入下面的命令：

```
$ make
```

Perl 的安装需要花费相当长的时间。当安装完成时，再键入下面这两个命令：

```
$ make test
# make install
```

make test 命令用于确保Perl的安装百分之百正确并且使之可以准备运行。若要运行 make install 命令，你必须以 root 身份进行登录（正因为这个原因，所以使用提示符#，这是根用户的提示符），因为它需要将 Perl 安装到系统目录中去。

当 make install 运行正确时，还可以测试 Perl 的安装情况，方法是再次键入下面的命令：

```
$ Perl –v
```

如果这个命令运行正确，那么安装成功了！



几乎在所有的 Linux 或者是 UNIX 程序中，你都会看到一个叫做 Makefile 的文件，它总是和 make 命令在一起使用。无论是在 Linux 还是在 UNIX 环境中，make 都是一个非常重要的编译命令，不管是自己进行项目开发还是安装应用软件，都经常要用到 make 或 make install。利用 make 工具，可以将大型的开发项目分解成为多个更易于管理的模块，对于一个包括几百个源文件的应用程序，使用 make 和 makefile 工具就可以简洁明快地理顺各个源文件之间纷繁复杂的相互关系。所以建议读者花点时间去学习 make 和 makefile，掌握之后，就再也不会对 Linux 下的应用软件手足无措了。

### 1.2.4 Perl 资源

网络上有许多关于 Perl 的资源，读者可以在表 1-1 和表 1-2 中列出的 Web 站点上找到你所想要的东西。

表 1-1　　北美的 Perl 资源一览表

| Site | Location | | |
|---|---|---|---|
| ftp.netlabs.com | IP 地址： | 192.94.48.152 | 目录 /pub/outgoing/Perl5.0 |
| ftp.cis.ufl.edu | IP 地址： | 128.227.100.198 | 目录 /pub/Perl/src/5.0 |
| ftp.uu.net | IP 地址： | 192.48.96.9 | 目录 /languages/Perl |
| ftp.khoros.unm.edu | IP 地址： | 198.59.155.28 | 目录 /pub/Perl |
| ftp.cbi.tamucc.edu | IP 地址： | 165.95.1.3 | |
| ftp.metronet.com | IP 地址： | 192.245.137.1 | 目录 /pub/Perl/sources |
| genetics.upenn.edu | IP 地址： | 128.91.200.37 | 目录 /Perl5 |

表 1-2 欧洲的 Perl 资源一览表

| Site | Location | | |
|---|---|---|---|
| ftp.cs.ruu.nl | IP 地址： | 131.211.80.17 | 目录 /pub/PERL/Perl5.0/src |
| ftp.funet.fi | IP 地址： | 128.214.248.6 | 目录 /pub/languages/Perl/ports/Perl5 |
| ftp.zrz.tu-berlin.de | IP 地址： | 130.149.4.40 | 目录 /pub/UNIX/Perl |
| src.doc.ic.ac.uk | IP 地址： | 146.169.17.5 | 目录 /packages/Perl5 |

## 1.3 小结与练习

### 1.3.1 小结

本章简单地介绍了 Linux 操作系统和 Perl 编程语言的历史。Linux 系统和 Perl 语言都有着自己的文化，希望读者能够在此看到其文化发展的根源。此外还对 Linux 系统的版本和使用现状作了介绍，使读者对其有所了解，同时还介绍了 Perl 的安装，这样可使读者对 Perl 编程有了一个感性上的初步认识。在本章最后所附的 Perl 的资源列表中，读者可以找到大量与 Perl 相关的资源。

### 1.3.2 习题与思考

设计一个方案，以便有效地学习 Linux 下的 Perl 编程。

# 第2章

# 标量数据

标量变量简介
数字标量
字符串标量
常用的与标量有关的系统函数
小结与练习



数据结构的简单性是 Perl 5 的优点之一。许多语言对建立数据结构指定了大量条件，使得编程工作非常繁琐。Perl 5 的内置函数能操作数据，对建立数据结构指定条件极少，数据可以在程序中任何地方生成。在 Perl 中只需考虑 3 种基本数据结构：标量、数组和散列。本章重点介绍标量数据。

本章要点：

- 数字型标量。
- 标量的命名法则。
- 数字型标量的几种表示法。
- 3 种引号对字符串的作用。
- 常用的与标量有关的系统函数。

# 2.1 标量变量简介

## 2.1.1 何谓标量变量

到底什么是标量变量，下面先给出一个简单的例子，让读者有一个感性认识。

```
$Name = "Tom";
print "My name is $Name";
```

在这个例子中，$Name 就是一个标量变量。这两个语句的执行结果是在屏幕上打印出“My name is Tom”。

标量（scalar）是 Perl 里最单纯的数据类型。变量是可修改的内存地址，用于保存信息以备今后使用。所谓标量变量，顾名思义，就是一种用来存放标量的变量。

Perl 中有两种标量数据：

- 数字，如 24、5.76、3.25e20 等；
- 字符串，如“How are you!”等。

读者可能会问，一个变量怎么可以既用来存放数字，又用来存放字符串呢？的确，这与以前的经验有很大不同。但 Perl 确实能做到，它把它们视为可相互替代的东西，就像一个抽屉里既可以放书也可以放衣服。这使我们的程序大大地简化了，也增加了系统的灵活性。

标量值可用操作符运算，结果通常也是标量值。标量值可用标量变量加以储存，也可以从文件或其他装置读入或写出。

关于标量变量，记住下列 3 条规则：

- 标量变量一次只能存放 1 个数据项。
- Perl 5 标量变量可以包含数字或字符串数据。
- Perl 5 根据上下文确定标量变量存放数字或字符串数据。Perl 5 根据操作符、数据类

型和变量本身确定上下文。如：操作符 eq 表示对字符串数据操作；操作符 == 表示对数字进行操作。数据（如 67.09 或“Stnng”）也帮助变量确定上下文。

### 2.1.2　标量变量的命名

前面提过的标量名称$Name 符合 Perl 5 的标量变量命名规则。Perl 5 允许在标量变量中使用符合规则的字母、数字或非字母数字字符。

所有标量变量名都必须以$号开头，$号后面的字符统称为变量名。下面是 Perl 5 的变量命名规则：

- 以下划线或字母开头的变量名后面的字符可以是数字、下划线或字母，其长度不限（有些版本将变量名长度限制为 255 个字符）。
- 以数字（0~9）开头的变量名可以是任意长度，但只能包含数字，如$982。
- 变量名不能包含空格。
- 变量名可以包含任何可打印字符（除空格），但以非字母数字字符开头的变量名只能包含一个字符。



$ 号是 Perl 变量指定符，指定变量类型为标量。Perl 5 也用@号表示数组变量名，用%号表示散列变量名。大多数 Perl 编程人员把变量类型指定符看作变量名的一部分。Perl 5 的许多内部特殊变量只能包含一个字符。例如，进程 ID 的变量名为 $!。

## 2.2　数字标量

所谓数字标量，顾名思义，就是数字型的标量，这与我们在其他语言中看到的基本相同，下面就是一些数字标量的例子：

```
12345        # 正整数
54321        # 负整数
12345.67     # 浮点数
6.02E23      # 科学计数法表示的数
0xffff       # 十六进制形式表示的整数
0377         # 八进制形式表示的整数
```

从上面的这些例子中不难看出，数字有多种表示类型，通常将它们分为整数和浮点数。不过，在 Perl 的内部只有一种数字类型，就是双精度浮点数，即使整数也会用等价的浮点数来表示，整数实为浮点数的特例。从程序设计的观点来看，这倒没有什么影响。下面将具体介绍一下数字的整型表示法和浮点型表示法。

### 2.2.1 整型表示法

整型是 Perl 里最常用的简单变量，这与其他语言相同。下面分别介绍整数的十进制、八进制及十六进制表示法。

1. 十进制整数

十进制整数表示法是最常见的一种整数表示法，例如：

```
112
+5
2004
3485
```

下面是一个应用了十进制整数的例子：

**程序清单 2-1：十进制整数的应用**

```
$x = 345;
$y = -678;
if (89 + 916 == 7333){
      print $x;
}
else {
      print $y;
}
```

该程序比较简单，它首先声明了两个标量变量$x 和$y，并给它们赋了初值。然后判断 89 与 916 的和是否等于 7333，如果等于，则输出$x 的值，否则输出$y 的值。程序中 345，89 就是一个整型十进制直接数。

该程序的执行结果是在屏幕上输出：

```
618
```



所谓直接数，是指源代码中的常量值，直接数与变量恰好相反，不能修改。变量包含数值，可以修改再包含不同的值。如 90 和 34.76 是数字直接数，“Hello” 是字符串直接数。

2. 八进制和十六进制数

讲解八进制和十六进制之前，有必要先介绍一下二进制。

大家知道，从条件表达式（如 if ($a = = $b)）返回的布尔值仅为真或假。真或假是相对的，就像开和关。

二进制操作涉及 0 或 1，所以要了解二进制操作，必须先了解二进制数及 0 和 1。每个

二进制位表示计算机内存中的一位。计算机只知道数字，实际上，它也只认识 1 和 0。计算机所做的一切就是用布尔 1 和 0 的组合，构成二进制的整个数字集。计算机将所有字母和数字变成二进制 1 和 0。二进制 10 等于十进制 2，二进制 110 等于十进制 4 加 2 即 6。

将二进制数 3 位一组进行组合，就可以得到八进制数。八进制的基数为 8，即每位表示 8 个数之一，八进制采用的八个数字为 0 到 7。

将二进制数 4 位一组进行组合，就可以得到十六进制数。十六进制的基数为 16，即每位表示 16 个数之一。十六进制采用的前十个数字与十进制一样为 0 到 9，另外的六个取英文中的前六个字母 A 到 F。这样，0 到 F 就可以表示十进制值 0 到 15。

表 2-1 给出了 0 到 15 的十进制、二进制、八进制和十六进制表示。

表 2-1　　0-15 的十进制、二进制、八进制和十六进制表示对比

| 十进制数 | 二进制数 | 八进制数 | 十六进制 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 |
| 3 | 11 | 3 | 3 |
| 4 | 100 | 4 | 4 |
| 5 | 101 | 5 | 5 |
| 6 | 110 | 6 | 6 |
| 7 | 111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |

需要特别指出的是，Perl 5 可以表示八进制数、十进制数和十六进制数，但没有办法表示二进制数。也就是说，使用二进制操作符时要通过心算把一种进制换算成二进制，从而确定结果。建议二进制操作中尽量用十六进制。

在 Perl 5 中，八进制以 0 开头，十六进制以 0x 开头。例如：

```
$var1 = 047; # 等于十进制的39
```

```
$var2 = 0x1f;# 等于十进制的31
```

3. 将浮点数变成整数

处理数字问题时，有时需要将浮点数变成整数，这可以由函数 int()来实现，下面是一个例子：

**程序清单 2-2：将浮点数变成整数**

```
$average_age1 = 939/16;
print $average_age1;        # 打印结果为58.6875
$average_age2 = int 939/16;
print $average_age2;        # 打印结果为58
```

## 2.2.2 浮点型表示法

Perl 接受所有 C 语言的浮点表示法。数字可以有小数点，也可以没有；数字的前面也可以加上正号或负号。另外，我们也可以用“E”或“e”表示某数再乘上 10 的几次方。 下面是浮点型表示的一些例子：

```
1.25
7.25e45
6.5e24
12e-24
1.  2E-23
```

浮点寄存器通常不能精确地存储浮点数，从而产生误差，在运算和比较中要特别注意。指数的范围通常为-309 到+308。这一点可由下面的例子加以说明：

**程序清单 2-3：浮点数计算的误差**

```
$value = 9.01e+39 + 0.01 - 9.01e+39;
print ("The first value is ", $value, "\n");
$value = 9.01e+39 - 9.01e+39 + 0.01;
print ("The second value is ", $value, "\n");
```

程序执行结果为在屏幕上打印出：

```
The first value is 0
The second value is 0.01
```

### 2.2.3　数字格式

Perl 用标准数学符号定义数字直接数，见表 2-2。

表 2-2　　数字直接数的格式与记法

| 类　型 | 语　法 | 示　例 |
|---|---|---|
| 整型 | NN | 89 |
| 浮点数 | NN.NN | 12.34 |
| 科学计数 | NN.NNENN | 23.04E-5 |
| 大数 | NN_NNN_NNN | 19_000_000 |
| 八进制 | 0NNN | 0761 |
| 十六进制 | 0NNNN | 0xF9DA |

### 2.2.4　定点数方案

Perl 5 能够大大简化我们的工作，但它在数字方面仍然有一些不足，比如，它没有定义舍入、截尾和定点数计算的方法。这对涉及货币的计算和需要定点数的应用程序就会出问题。

Perl 5 中涉及到定点数时通常有以下 3 种方法：

- 用 sprintf 函数将结果舍入为定点数。
- 生成舍入函数，产生可以控制的舍入值。
- 用正则表达式将结果截尾为定点数。

这里，我们只对第一种方案加以说明，其他两种方案大家在学习了 Perl 5 的函数和正则表达式后，可以参阅其他资料。

下面介绍 sprintf 函数的使用。

sprintf 函数是用于格式化打印数据和其他显示功能的函数，其名称的含义是字符串打印格式（string print format）。该函数舍入成所请求的格式，它有两个输入参数：第一个是数据格式参数，第二个是要格式化的数据，函数返回经过格式化的结果。下面是一个例子：

```
$total = 18;
$roundedValue = sprintf ( " %0.2f", $total);
```

第一个参数" %0.2f"的解释如下：

- f 表示格式为浮点数。
- 2 表示取小数点后面两位。
- 小数点前面的 0 表示在空缺的位置添上 0，称为填充。

这样上例的操作结果为：

```
18.00
```

# 2.3 字符串标量

字符串是由一连串的字符组成的，每个字符以 8 比特（bit）的值表示。因此整个字符集共有 256 个字符。Perl 里面的 NULL 字符并没什么特殊之处，这与其他语言不同。

引号内的一切都被认为是字符串，因为它通常是由几个字符构成单字。Perl 5 中，一个字符和多个字符都称为字符串。最短的字符串甚至可以不包含任何字符，即空字符串。Perl 并未规定字符串的最大长度，内存有多大，字符串就可以有多长。Perl 很多地方都没有“内置的极限”，这是它的特点之一。字符串一般由“可打印的字符”组成，即 ASCII 编码的 23~126 之内的字符。其他字符虽然不可打印，但常会在处理二进制数据时遇到。

Perl 允许多种字符串表示法，其中最常用的两种是“单引号字符串”（single-quoted string）和“双引号字符串”（double-quoted string）；另外还有一种不常用的表示法称为“反引号字符串”（back-quoted string）。每种引号具有不同的意义，下面将分别介绍。

## 2.3.1 单引号字符串

顾名思义，单引号字符串就是一前一后用单引号括起来的字符串。单引号本身并不是字符串的一部分，只是用它来标示出字符串的开头和结尾。两个单引号中间的任何字符（若字符串以一行以上表示，则还包括换行字符）都被视为合法。若想在单引号字符串里再表示单引号（'），必须在引号前面加反斜线（backslash，\）；若想表示反斜线，要在前面再加一个反斜线。请看下面的例子：

```
'work'     #  4个字符：w，o，r，k
'are\'t'    #  3个字符：a，r，e，单引号，t
''     #  空字符串 (不含任何字符)
'follow\\me'     #  f，o，l，l，o，w，\，m，e
'hello\n'     #  hello后面接反斜线再接n
'hello
there'     # hello后面跟一个换行符，再接there (一共11个字符)
```

注意，这里\n 并不解释为换行字符。反斜线必须在另一个反斜线或单引号的后面才代表特别的意义。

从上面的例子我们还可以看出，单引号内是可以跨多行的，如：

```
$text = 'This is two
```

```
lines of text
';
```

与下句等效：

```
$text = "This is two\nlines of text\n";
```



单引号字符串与双引号字符串（下面将要介绍）有两个区别：一是没有变量替换功能，二是反斜线不支持转义字符，而只在包含单引号和反斜线时起作用。

### 2.3.2　双引号字符串

Perl 的双引号字符串和 C 语言类似，形式上都是“引号+字符串内容+引号”。它和单引号字符串不同的是，反斜线在这里具有完全的控制字符能力，可以支持转移字符和变量替换，甚至可以用它表示任何八进制或十六进制的字符。

```
"hello world\n"     # hello world, 和一个换行符
"new \007"          # new, space,和一个响铃符 (八进制007)
"coke\tsprite"      # coke, 一个制表符,和sprite
"c:\\temp"          # c:, 反斜杠, 和temp
```

1. 双引号字符串中的转移字符

反斜线可以加在许多字符前面，分别代表不同的意义。表 2-3 是它完整的列表。

表 2-3　　双引号字符串中的转义字符

| 转移字符 | 意义描述 |
|---|---|
| \a | 响铃(Bell) |
| \b | 退格(Backspace) |
| \cn | Ctrl+n 作控制字符 |
| \e | Escape |
| \f | Formfeed |
| \n | 换行（Newline） |
| \t | Tab |

续表

| 转移字符 | 意义描述 |
|---|---|
| \r | 回车(return) |
| \l | 下个字符小写 |
| \L | 由下个字符开始，到\E 为止的字符一律小写 |
| \u | 下个字符大写 |
| \U | 由下个字符开始，到\E 为止的字符一律大写 |
| \Q | 由下个字符到\E 为止的所有非字母或数字的字符前面全部加上反斜线 |
| \E | 结束\L，\U 或\Q 的作用 |
| \007 | 任何八进制 ASCII 值 |

下面是几个例子：

**程序清单 2-4：使用双引号转义字符**

```
print "hello world\n"      #  hello，空白，world和换行
print "ride\tbike"         #  ride，  tab字符和bike
print "T\LHIS IS A \ESTRING";     #  This is a STRING
print "Bill of Goods
Bread:\t\$34.45\n";
print "Fruit:\t";
print "\$45.00\n";
print "\t======\n";
print "\t\$79.45\n";
```

程序输出结果为：

```
hello world
ride    bikeThis is a STRINGBill of Goods
Bread:  $34.45
Fruit:  $45.00
        ======
        $79.45
```

下面是利用表 2-3 中的转移字符改变字符串大小写的一个例子：

程序清单 2-5：改变字符串的大小写

```
$bigfred = "\Ufred";
print $bigfred,"\n";
$fred = "fred";
$bigfred = "\U$fred";
print $bigfred,"\n";
$capfred = "\u$fred";
print $capfred,"\n";
$barney = "\LBARNEY";
print $barney,"\n";
$capbarney = "\u\LBARNEY";
print $capbarney,"\n";
$bigbarney = "BARNEY";
print $bigbarney,"\n";
$capbarney = "\u\L$bigbarney";
print $capbarney,"\n";
```

程序输出结果为：

```
FRED
FRED
Fred
barney
Barney
BARNEY
Barney
```

要在字符串中包含双引号或反斜线，则在其前加一个反斜线，如：

```
$res = "A quote \" and A backslash \\";
```

2. 双引号字符串中的变量替换

前面已经提到，双引号内的字符串中支持简单变量替换，例如：

```
$number = 9801;
$text = "The text contains a number: $number.
```

则$text 的内容为：

"The text contains a number: 9801."

下面这段程序可使读者更好地理解字符串中的变量替换：

**程序清单 2-6：字符串中的变量替换**

```
$fred = "pay";
$fredday = "wrong!";
$barney = "It's $fredday";     #  $barney不是"It's payday!", 而是 "It's wrong!"
$barney = "It's ${fred}day";     #  现在$barney是 "It's payday"
$barney2 = "It's $fred"."day";   # 同上
$barney3 = "It's " . $fred . "day";     # 同上
```

双引号内的反斜线可以取消变量替换，如：

```
$result = 14;
print ("The value of \$result is $result.\n"); # 执行结果：The value of $result is 14.
```

3. 双引号字符串中对八进制和十六进制的支持

可用\nnn（八进制）或\xnn（十六进制）来表示ASCII字符，如：

```
"new\177"     #  new，空白和delete字符 （编号八进制177的字符）
$result = "\377"; #  表示ASCII码值为255的字符，为EOF
 $result = "\xff";        #  也表示ASCII码值为255的字符
```



习惯使用 C 语言的程序员要注意，在 Perl 中，字符串的末尾并没有隐含的 NULL 字符，NULL 字符可以出现在串的任何位置。

### 2.3.3 反引号

反引号（`）让 Perl 5 把引号之间的字符串解释为操作系统命令。如果要在一个程序中执行另一个程序，则可以使用反引号。反引号间执行的操作系统命令总是返回某种类型的数据。对有些命令，系统只能返回表示成功与否的 0 和 1 值。对某些命令，则可能返回长长的数据项表。返回的数据存放在等号左边的变量中。下面的例子展示了如何使用反引号：

**程序清单 2-7：反引号的使用**

```
$directory = `dir/w`;
```

```
@directory = 'dir/w';
print "Scalar context directory listing\n $directory\n";
print "List context directory listing\n @directory\n";
```

### 2.3.4　引用操作符

如果对引号操作符感到费解或是想在屏幕上打印引号，可以用 Perl 5 提供的 4 种引用操作符：q、qq、qx 和 qw。这些引用操作符可以替代相应的引号，见表 2-4。

表 2-4　　引用操作符

| 操作符 | 含　义 | 结　果 |
|---|---|---|
| q | 单引号 | 不可插值字符串 |
| qq | 双引号 | 可插值字符串 |
| qx | 反引号 | 可插值命令字符集 |
| qw | 引用字符串 | 不可插值字符串 |

引用操作符的用法很简单。每个引用操作符作用于分割符间的字符，引用操作符的分割符是引用操作符后面的第一个非字母或数字字符，空格不能做操作符。引用操作符应成对出现，结束分割符应与开始分割符相同。分割符可以用任何非字母或数字字符对。分割符是表示语句开始和结束的任何字符。

## 2.4　常用的与标量有关的系统函数

下面将向读者介绍一些常用的系统内置函数，它们通常都以常量或标量为参数，在后面的章节将会经常用到，所以在这里分门别类地介绍它们。

### 2.4.1　数学函数

表 2-5 是 Perl 中常用的数学函数，用法与其他语言中的大致相同：

表 2-5　　常用的数学函数

| 函数名 | 调用语法 | 返回值 | 说明 |
|---|---|---|---|
| sin | retval = sin (value) | 计算结果 | 参数为弧度值 |
| cos | retval = cos (value) | 计算结果 | 参数为弧度值 |
| atan2 | retval = atan2 (value1, value2) | 运算返回 value1 除以 value2 结果的 arctan 值 | 单位为弧度，范围在-PI~PI |

续表

| 函数名 | 调用语法 | 返回值 | 说明 |
|---|---|---|---|
| sqrt | retval = sqrt (value) | 计算结果 | value 为非负数 |
| exp | retval = exp (value) | 返回 e 的 value 次方 | |
| log | retval = log (value) | 以 e 为底的自然对数 | value 为正数 |
| abs | retval = abs (value) | value 的绝对值 | |
| rand | retval = rand (num) | 返回 0 和整数 num 之间的一个浮点数 | 随机数函数 |
| srand | srand (value) | 无 | 初始化随机数生成器，保证每次调用 rand 真正随机 |

## 2.4.2 标量转换函数

表 2-6 是 Perl 中常用的标量转换函数：

表 2-6 常用的标量转换函数

| 函数名 | 调用语法 | 返回值 | 说明 |
|---|---|---|---|
| chop | $lastchar = chop (var) | 当 var 为变量时，返回最后一个字符；当 var 为数组/列表时，所有元素的最后一个字符被删除，并返回最后一个元素的最后一个字母 | 该函数删除参数 var 的最后一个字符。var 必须为变量，因为它的值会改变 |
| chomp | result = chomp(var) | 返回值为实际删除的字符个数 | 检查字符串或字符串列表中元素的最后一个字符是否为由系统变量$/定义的行分隔符，如果是就删除 |
| crypt | result = crypt (original, salt) | 返回值为加密后的串 | 用 DES 算法加密字符串，original 是将要加密的字符串；salt 是两个字符的字符串，定义如何改变 DES 算法，以使更难于解码 |
| hex | decnum=hex(hexnum) | 转换后的十进制数 | 将十六进制数(字符串形式)转化为十进制数 |
| int | intnum=int (floatnum) | 转换后的整型数 | 将浮点数舍去小数部分转化为整型数 |

续表

| 函数名 | 调用语法 | 返回值 | 说明 |
| --- | --- | --- | --- |
| oct | decnum=oct (octnum) | 转换后的十进制数 | 将八进制数(字符串形式)或十六进制数(“0x..”形式)转化为十进制数 |
| pack | formatstr=pack(packformat, list) | 变换后的字符串 | 取一个或几个字符串并将对应字符变为指定格式（格式字符见附录二） |
| unpack | @list=unpack(packformat, formatstr) | 变换后的列表 | unpack 与 pack 功能相反，将以机器格式存储的值转化成Perl中值的列表。其格式字符与 pack 基本相同 |
| vec | retval = vec (vector, index, bits) | 返回所获得的块数据 | 矢量函数，它把简单变量 vector 的值看作多块(维)数据，每块含一定数目的位，合起来即一个矢量数据。参数 index 指出访问哪一块，访问次序是从右到左的；参数 bits 指定每块中的位数，可以为 1,2,4,8,16 或 32 |
| defined | retval = defined (expr) | 如果已定义，返回真；否则返回假 | 判断一个变量、数组或数组的一个元素是否已经被赋值。expr 为变量名、数组名或一个数组元素 |
| undef | retval = undef (expr) | 返回值始终为未定义值，此值与空串等效 | 取消变量、数组或数组元素甚至子程序的定义，回收其空间 |

## 2.5　小结与练习

### 2.5.1　小结

本章介绍了标量变量，它是 Perl 5 的主要数据类型之一。

第 1 节首先介绍了标量变量的定义，即变量是可修改的内存地址，标量变量是只包含一个项目的变量。在 Perl 5 中，这个项目可以是数字值或字符串。然后本节还介绍了标量变量的命名规则。

第 2 节介绍了两种标量变量之一的数字标量。数字标量可以是整型也可以是浮点型。整型既可以用十进制表示，也可以用八进制和十六进制表示。浮点数可以表示非整型数值，科

学计数法就是一种浮点数表示方法。接着我们还讨论了数字的格式和定点数方案。

第 3 节介绍了另一种标量变量——字符串标量。字符串又分为单引号字符串和双引号字符串，它们之间是有区别的。接着我们讨论了两个主题——反引号和引用操作符，它们都是为了使程序更加简单易读。

本章最后一节介绍了一些常用的与标量有关的系统内置函数，包括数学函数和标量转换函数。

通过学习本章，读者应该对 Perl 中的标量有了一定的理解，并开始初步掌握其用法。下一章将重点介绍 Perl 的第二种基本数据结构——数组与列表。

### 2.5.2 习题与思考

（1）写出下列数所表示的十进制数。

```
2.33e3
0.673e-2
0377
0xff
```

（2）写出下列各项所表示的字符串。

```
'hello'
'don\'t'
''
'silly\\me'
"\nhow are you\n"
"all\tright"
 "d:\\perl"
```

（3）分别写出下列各项中等式左边变量的值。

```
$a = "fred";
$b = "some text $a";
$c = "no such variable $what";
$x = '$fred';
$y = "hey $x";
$fred     = 'hi';
$barney   = "a test of \$fred";
```

（4）求一个半径为 7.6 的圆的周长。

# 第3章

# 数组和列表数据

数组和列表的基本概念
数组元素的存取
对数组元素的其他操作
与数组有关的函数
小结与练习



上一章介绍了标量数据，本章将介绍 Perl 的另一种数据类型：数组。如果把标量比做放一双鞋的鞋盒，则数组可以比做鞋柜中的鞋架，它可以放许多双鞋，即数组可以放置许多标量变量。

本章要点：

- 存取数组中的元素。
- 引号内使用数组及其转义。
- 与数组操作有关的函数。

# 3.1 数组和列表的基本概念

## 3.1.1 什么是数组和列表

所谓列表（list），就是按照次序排列的一连串的标量数据；数组（array）则是用来储存一个列表的变量。与我们在 C 语言中见到的数组不同，Perl 数组中的每个元素都是一个独立的标量变量，各自储存互不相干的标量值。这些标量值有顺序之分，也就是说，数组中的元素按照一个特定的次序，由最低位元素排列到最高位元素。列表可以有任何数量的元素：最小的列表不包含任何元素，最大的列表可以把任何计算机的内存塞满。

## 3.1.2 数组的表示

列表的表示法很简单，只要在一对小括号里面写入想要的值，它们之间用逗号隔开即可。也就是说列表是包含在括号里的一序列的值，可以为任何数值，也可为空，如：(1,5.3,"hello",2)，空列表：()。注意，只含有一个数值的列表 [如：(123)] 与该数值本身（即：123）是不同的，但它们可以互相转化或赋值。

下面是几个列表的例子：

```
(17,$var,"astring")
(17,26<<2)
(17,$var1+$var2)
($value,"Theansweris$value")
```

从中可以看出，列表元素不一定是常数，它可以是变量，它们会在程序执行时换算成具体的值。

### 1. 数组建构操作符

在表示数组的小括号里，还可以置入数组建构操作符（..），代表两个值中间的所有整

数。该操作符将产生开始于左运算元，以 1 递增，终止于右运算元的一连串整数。请看下面的例子：

```
(1..10)=(1,2,3,4,5,6,7,8,9,10)
(2,5..7,11)=(2,5,6,7,11)
(3..3)=(3)
```

如果右运算元比左运算元小，会得到空字符串。如果右运算元和左运算元的差不是整数，数组建构操作符会让产生的最后一个数字落在范围之内。例如：

```
(4.5..1.6)=()
(2.1..5.3)=(2.1,3.1,4.1,5.1)
```

数组建构操作符还可用于字符串：

```
("aaa".."aad")=("aaa","aab","aac","aad")
@day_of_month=("01".."31");
```

数组建构操作符也可包含变量或表达式：

```
($var1..$var2+5)     #依$var1和$var2当时的值而定
```

2. 字符串作数组数据

数组数据也可以是字符串，不过在小括号里加那么多括号，不仅难看而且很麻烦：

```
@a=("apple","orange","pear","banana","peach");
```

比较好的写法是用 qw 这个函数产生上面这样的数组，该函数会用小括号内非空白的部分，产生字符串数组数据，像下面这样：

```
@a=qw(appleorangepearbananapeach);     #这种写法较好
@a=qw(
apple
orange
pear
banana
peach
```

```
);    #效果与前一种一样，但显得很啰嗦
```

读者经常用到的 print 这个函数，它的参数其实也是数组：

```
print("Thenumberis",$a,"\n");
```

该函数会把参数的内容原原本本地打印在屏幕上。
下面是使用输出函数的一个小技巧：

**程序清单 3-1：使用输出函数的技巧**

```
$fred="Fred";
print(("Hello,".$fred."!\n")x2);
```

其结果是在屏幕上打印出：

```
Hello,Fred!
Hello,Fred!
```

其中用到的有关的操作符，我们将在后面的章节介绍。

### 3.1.3 列表变量

列表变量可以储存一个数组值。列表变量的命名法和标量变量很像，差别只在列表变量的名称必须以“@”符号开头，例如上面程序中的@a。值得注意的是，列表变量@a 和标量变量$a 没有任何关系，虽然它们名字相同。对于不同型态的变量，Perl 会分别维护不同的命名空间。

列表变量被赋值前，其值为空数组。整个列表本身，或其中的元素，都可以是合法的表达式。

下面是几个全局的数组变量，也就是说它们是系统定义的。

- ARGV—包含命令行变元的数组。
- INC—查找 doEXPR、require 或 use 结构求值的 Perl 脚本所在的地址表数组。
- @F—当命令行中用了–a 参数时，输入行被拆分后装入到其中的数组。

## 3.2　数组元素的存取

### 3.2.1　列表的赋值

在进行存取运算之前，必须先给列表赋值。列表赋值操作符是一个等号（=），和标量赋值操作符一样。我们不必担心系统会弄混，事实上，Perl 会根据变量的形态，自动判定应该用哪一个操作符。请看下面的例子：

```
@a=(1,2,3);
@b=@a;     #把@a的值拷贝给@b
```

如果把一个单独的标量值赋给列表变量，它会成为列表的一个元素：

```
@a=23;     #此时23会自动转变为（23）
```

用户也可以在数组中加入列表变量。程序运行到这样的数组时，会自动把变量替换成列表目前的值，像这样：

```
@a=qw(appleorange);
@b=(4,5,@a,6,7);          #@b现在是(4,5，"apple","orange"，6，7)
@b=(9，@b);     #@b现在是(9,4,5，"apple","orange"，6，7)
@b=(@b,"pear");      #@b现在是(9,4,5，"apple","orange"，6，7,"pear")
```

如果某数组所有的元素都是变量，此数组可被视为变量。也就是说，这样的数组可以放在等号左边，其中每个元素会被设成等号右边数组中对应位置的值。下面是一个例子：

```
($a,$b,$c)=(1,2,3);     #现在$a=1,$b=2,$c=3
($a,$b)=($b,$a);     #将$a和$b的值互换
($d,@e)=($a,$b,$c);     #把$a的值赋给$d，@e设成($b，$c)
($f,@e)=(@e);     #把@e的第一个元素赋给$f，即把$b的值赋给$f，@e=($c)
```

如果右边数组的元素个数超过左边的变量个数，是没什么关系的；但如果左边数组的变量个数超过右边的元素个数，多出来的变量值会是 undef。

数组变量若出现于数组中，它必须是最后一个元素，因为它会“吃掉”对应数组后面所有的值。如果有其他变量置于其后，它们的值会是 undef。

如果把列表变量赋值给标量变量，标量变量的值会被 Perl 解释为列表的长度，像这样：

```
@array=(1,2,3);
$scalar=@array;     #$scalar=3,即@array的长度
($scalar)=@array;     #$scalar=1,即@array第一个元素的值
$x=(($a,$b)=(7,7,7));     #$x=3，不是2
```

此外，还可以从标准输入<STDIN>给变量赋值。<STDIN>在标量环境下会传回输入的一行；但在列表环境下，会传回到文件结尾之前的所有行，每行都是数组中独立的一个元素。请看下例：

```
$var=<STDIN>;     #这里是标量环境，$var为输入的一行。
@array=<STDIN>;     #这里是列表环境，由标准输入设备读入数据
```

程序执行到上面的第二句时，若键入 3 行，然后再按下 CTRL-D（代表文件结束），数组 @array 会有 3 个元素，内容分别是输入的 3 行所组成的字符串（包括换行字符）。

### 3.2.2 存取数组元素

1. 以整数作下标的存取

对数组中的值通过下标存取，第一个元素下标为 0。注意在这里用“$”而不用“@”，因为列表中的某个元素是独立的变量。下面是一个例子：

```
@array=(1,2,3,4);
$scalar=$array[0];     #$scalar=1
```

因为数组中的元素是一个独立的变量，所以它可以放在“=”左边被赋值，如：

```
@array=(1,2,3,4);
$array[3]=5;     #现在@array=(1,2,3,5)
$array[3]++;     #现在@array=(1,2,3,6)
$array[3]+=3;     #现在@array=(1,2,3,9)
```

试图访问不存在的数组元素，则结果为 NULL，但如果给超出数组大小的元素赋值，则数组自动增长，原来没有的元素值为 NULL。如：

```
@array=(1,2,3,4);
$scalar=$array[4];     #$scalar=null;
$array[6]=7;     #现在@array=(1,2,3,4,"","",7)
```

除此之外，还可以进行数组间的整体拷贝：

```
@result=@original;
```

或用数组给列表赋值，如：

```
@list1=(2,3,4);
@list2=(1,@list1,5);      #@list2=(1,2,3,4,5)
```

数组可以对简单变量赋值：

```
@array=(5,7,11);
($var1,$var2)=@array;      #$var1=5,$var2=7,11被忽略
($var1,$var2,$var3)=@array;      #$var1=5,$var2=7,$var3=""(null)
```

有时用户需要存取数组中连续几个元素，如下面的语句：

```
($array[0],$array[1])=($array[1],$array[0]);
```

这种用法称为“slice”，因为常用，所以 Perl 为这种用法提供了特别的表示法：

```
@array[0,1];      #与($array[0],$array[1])相同
@array[0,1]=@array[1,0];      #等价于($array[0],$array[1])=($array[1],$array[0])
@array[0,1,2]=@array[1,1,1];      #现在array数组前3个元素的值都等于原第二个元素的值。
@array[0,1]=(8,9);      #$array[0]和$array[1]分别被赋予8和9
```

负数下标对数组来说是从后面倒着数的意思，所以可以用$array[-1]取到最后一个元素的值。请看下面的例子：

```
@array=(6,7,8);
$a=$array[-1];      #$a=8
$b=$array[$a];      #$b=4;
```

用小于零的数字当下标且赋值，常常会发生严重错误，千万要避免。

2. 以表达式做下标的存取

上面这些例子的数组下标值都是整数，但实际上它也可以是任何能传回整数的合法表达式，如：

```
@array=(1,2,3,4);
$a=3;
$b=$array[$a];      #$b=4;
$b=$array[$a-2];    #$b=2;
($c)=(1,2,3,4)[$a-2];    #$c=2;
```

在 slice 里也可以这样用。但要注意的是，用在这里的表达式必须传回数组，不能是数值：

```
@array=(6,7,8);
@order=(2,1,0);
@reverse=@array[@order];    #现在@reverse=@array[2,1,0]，即@reverse=(8,7,6)
```

用“$#arrayName”表达式可以取得数组目前最后一个元素的下标值。你可改动这个值，让数组缩短或伸长。不过通常没有必要这么做，系统会自动负责这件事。

```
@array=(5,6,7,8,9);
$x=$#array;    #$c=4;
$y=$array[$#array];    #$d=9;
```

3. 子数组

可以对数组中的一部分进行操作，这部分叫做原数组的子数组。下面是一些例子：

```
@array=(1,2,3,4,5);
@subarray=@array[0,1];    #@subarray=(1,2)
@subarray2=@array[1..3];    #@subarray2=(2,3,4)
@array[0,1]=("string",46);    #现在@array=("string",46,3,4,5)
@array[0..3]=(11,22,33,44);    #现在@array=(11,22,33,44,5)
@array[1,2,3]=@array[3,2,4];    #现在@array=(11,44,33,5,5)
@array[0..2]=@array[3,4];    #现在@array=(5,5,"",5,5)
```

也可以用子数组的形式来交换元素：

```
@array[1,2]=@array[2,1];
```

## 3.3　对数组元素的其他操作

### 3.3.1　字符串中的方括号和变量替换

数组值也可以像标量变量一样，安插在双引号表示的字符串里面。待程序执行到该处时，会用数组中某元素的值来替换它。请看下面的例子：

```
@a=qw(appleorangepearbananapeach);
$position=3;
$x="Thisisa$a[2]";      #"Thisisapear"
$x="Thisisa@a[$position-1]";      #"Thisisapear"
```

下标符号内的表达式会被优先计算，用其值取得数组的元素。也可把整个数组插入字符串，系统把数组所有元素的值替换进来（按下标顺序、中间隔出空白），用法如下：

```
@a=qw(appleorangepearbananapeach);
$fruit="@aareallfruit";      #"appleorangepearbananapeachareallfruit."
```

也可以在字符串中使用转义字符：
"$var\[0]"将字符"["转义，等价于"$var"."[0]"，$var 被变量替换，[0]保持不变。
"${var}[0]"亦等价于"$var"."[0]"。
"$\{var}"则取消了大括号的变量替换功能，包含文字：${var}.

### 3.3.2　数组的输出

可以用 print 函数输出数组，不用将每个数组元素单独输出，该函数可以将数组中的元素按顺序全部输出，用法如下：

```
print@arrayname;
```

下面是一个例子。

**程序清单 3-2：数组的输出**

```
@smallArrayOne=(5..10);
@smallArrayTwo=(1..5);
```

```
@largeArray=(@smallArrayOne,@smallArrayTwo);
print@largeArray;
```

该段代码执行的结果是在屏幕上输出：

```
567891012345
```

下面是另外一种输出数组的形式，这种形式中没有显式的给出数组变量。

**程序清单 3-3：数组输出的另一种形式**

```
print"Hereisanemptyarray:".()."<--Nothingthere!\n";
print(12,014,0x0c,34.34,23.3E-3);
print"\n";
print("This","is",'an',"array",'of',"strings");
print"\n";
print("This",30,"is",'a',"mixedarray",'of',0x08,"items");.
```

该段代码执行的结果是在屏幕上输出：

```
Hereisanemptyarray:<--Nothingthere!
12121234.340.0233
Thisisanarrayofstrings
This30isamixedarrayof8items
```

### 3.3.3 改变数组元素的值

当我们需要改变数组元素的值时，如果要改变的只是个别元素的值，我们可以将它们单独列出来，赋予新的值。

**程序清单 3-4：改变数组元素的值**

```
@array=(1..30);     #@array中的元素默认为0
print@array;
$array[2]=5;
$array[20]=25;
print@array;
```

但是如果要改变数组中大量的数据，这种单个的处理方法显然就不行了，这时可以利用下面的两种方法。

1. 方法一

用一个 for 或 while 循环对每个数组元素重新赋值。

（1）用 for 循环编程

**程序清单 3-5：用 for 循环改变数组元素的值**

```
@array=(1..30);
print@array;
for($i=0;$i<30;$i++){
    $array$i]=$array[$i]+10;     #将每个元素加10
}
print"@array\n";
```

当知道数组的长度时可用这种方法。

（2）以数组的长度为循环次数，用 while 循环可编程

**程序清单 3-6：用 while 循环改变数组元素的值**

```
@array=(1..30);
  $count=1;
  while($count<=@array){
        print("element$count:$array[$count-1]\n");
        $count++;
  }
```

2. 方法二

利用 foreach 函数，这样处理起来最为方便，程序也特别简洁。

**程序清单 3-7：用 foreach 函数改变数组元素的值**

```
@array=(1..30);
print@array;
foreach$item(@array){
    $item+=10;     #将每个元素加10
}
print"@array\n";
```

值得一提的是，方法二还可以对多个数组同时进行操作，这更加体现了它的优越性。

**程序清单 3-8：用 foreach 函数操作多个数组**

```
@a=(1,-7,0.5,3,34);
```

```
@b=(0,1,-3,0.46e2,3.45);
foreach$item(@a,@b){
    $item*=7;
}
print"@a@b\n";
```

打印结果为：

```
7-493.52123807-2132224.15
```

## 3.4 与数组有关的函数

本节将介绍与数组有关的一些函数，首先列出的是常用函数及其调用语法，然后针对其中一些经常要用到的函数给出了具体的例子，以便读者熟练掌握。

表 3-1 列出了常用的数组和列表函数：

表 3-1 常用的数组和列表函数

| 函数名 | 调用语法 | 返回值 | 说明 |
|---|---|---|---|
| grep | @foundlist=grep(pattern,@searchlist); | 返回值是匹配元素的列表 | 与同名的 UNIX 查找工具类似，grep 函数在列表中抽取与指定模式匹配的元素，参数 pattern 为欲查找的模式 |
| splice | @retval=splice(@array,skipelements,length,@newlist); | 返回被删除元素的列表 | 拼接函数可以向列表（数组）中间插入元素、删除子列表或替换子列表。参数 skipelements 是拼接前跳过的元素数目，length 是被替换的元素数，newlist 是将要拼接进来的列表 |
| shift | element=shift(@arrayvar); | 返回被删除的元素 | 删去数组第一个元素，剩下元素前移。不加参数时，缺省地对@ARGV 进行操作 |
| unshift | count=unshift(@arrayver,elements); | 返回值为结果(列表)的长度 | 作用与 shift 相反，在数组 arrayvar 开头增加一个或多个元素，等同于 splice(@array,0,0,elements) |
| push | push(@arrayvar,elements); | 返回新数组的长度 | 在数组末尾增加一个或多个元素，等同于 slice(@array,@array,0,elements) |

续表

| 函数名 | 调用语法 | 返回值 | 说明 |
| --- | --- | --- | --- |
| pop | element=pop(@arrayvar); | 返回值为删除的元素，当列表已空，则返回“未定义值”(即空串) | 与 push 作用相反，删除列表最后一个元素 |
| split | @list=split(pattern,string,maxlength); | 返回分割后的元素列表 | 将字符串分割成一组元素的列表。每匹配一次 pattern，就开始一个新元素，但 pattern 本身不包含在元素中。maxlength 是可选项，当指定它时，达到该长度就不再分割了 |
| sort | @sorted=sort(@list); | 排序后的列表 | 按字母顺序给列表排序 |
| reverse | @reversed=reverse(@list); | 排序后的列表 | 按字母逆序给列表排序 |
| map | @resultlist=map(expr,@list); | 返回值为计算后的结果列表 | 此函数在 Perl 5 中定义，可以把列表中的各个元素作为表达式 expr 的操作数进行运算，其本身不改变。在表达式 expr 中，系统变量$_代表各个元素 |
| wantarray | result=wantarray(); | 当子程序被期望返回列表时，此函数返回值为非零值(真)，否则为零值(假) | Perl 中，一些内置函数的行为根据其处理简单变量还是数组有所不同，如 chop。自定义的子程序也可以定义这样两种行为 |

下面举例详细介绍一下我们经常要用到的数组函数。

### 3.4.1　按字符顺序排序——sort 函数

函数 sort 会依 ASCII 码的顺序，对参数进行排序，以此结果产生新数组做为返回值，原参数不会被改变。请看下面的例子：

```
@array=("this","is","a","test");
@array2=sort(@array);      #@array2=("a","is","test","this")
@array=(70,100,8);
@array=sort(@array);       #@array=(100,70,8)
```

注意，此时结果并非数字的顺序，而是照字符串大小排序的。后面章节将会讨论如何按

数字大小以及其他条件的排序方法。

### 3.4.2 反转数组—— reverse 函数

reverse 函数以数组为参数，返回的值也是一个数组，内容是将参数内的元素次序反过来。请看下面的例子：

```
@a=(7,8,9);
@b=reverse(@a);     #@b的值是(9,8,7)，@a=(7,8,9)
@b=reverse(7,8,9);     #@b的值也是(9,8,7)
```

注意：reverse 函数并不会对参数进行任何改变。若想对参数进行反转，必须用函数的返回值设回来：

```
@a=(7,8,9);
@a=reverse(@a);     #@a的值是(9,8,7)
```

### 3.4.3 数组去尾——chop 和 chomp 函数

在前面讨论字符串变量时我们已经用过 chop 和 chomp 函数，它的参数也可以是数组变量。

chop 会删除每个元素最后的字符，见下面的例子：

```
@list=("rabbit","12345","quartz\n");
chop(@list);     #@list=("rabbi","1234","quartz")
```

而 chomp 会删除每个元素最后的分隔字符；如果没有分割字符，则不删除，见下面的例子：

```
@list=("rabbit\n","12345","quartz\n","fine");
chomp(@list);     #@list=("rabbit","12345","quartz","fine")
```

### 3.4.4 数组连接/拆分——join/split 函数

连接函数 join 的第一个参数是连接所用的中间字符，其余则为待连接的字符或字符数组。见下面的例子：

```
$string=join("","this","is","a","string");      #结果为"thisisastring"
@list=("words","and");
$string=join("::",@list,"colons");      #结果为"words::and::colons"
```

拆分函数 split 的第一个参数是拆分所用的中间字符，第二个则为待拆分的字符数组。见下面的例子：

```
$string="words::and::colons";
@array=split(/::/,$string);      #@array=("words","and","colons")
```

### 3.4.5　push 和 pop 函数

Perl 提供了 push 和 pop 这两个函数，用以把数组当作堆栈（stack）使用，以方便程序设计者进行堆栈的操作，因为堆栈是很常用的。push 和 pop 函数的用法与真正的堆栈操作基本一致。请看下面的例子：

```
push(@list,$value);      #相当于@list=(@list,$value)
$lastValue=pop(@list);      #删除@list最后一个元素并返回该元素
```

如果用空数组作为 pop 的参数，系统返回 undef，而不会产生任何警告信息。

push 函数可以一次对某数组附加一个以上的数值，它们会被按照次序接在数组的后面。请看下面例子：

```
@list=(10,20,30);
push(@list,40,50,60);      #@list=(10,20,30,40,50,60);
```

注意第一个参数必须是数组变量。

push 函数还可以将一个数组完整地加到另一个数组后面：

**程序清单 3-9：用 push 函数实现数组合并**

```
@members=("Time","Flies");
@initiates=("An","Arrow");
push(@members,@initiates);
print@members;
```

打印结果为：

```
TimeFliesAnArrow
```

### 3.4.6 shift 和 unshift 函数

上面的 push 和 pop 函数对数组的最右边作用，这里提供的 shift 和 unshift 函数则会对数组的最左边作用。shift 函数删除数组最左边的元素并返回该元素的值，相当于 pop 函数；unshift 将所给的数据添加到数组中作为数组的第一个元素，相当于 push 函数。请看下面的例子：

```
@arrat=(7,8,9);
$a=6;
unshift(@array,$a);      #@array=(6,7,8,9);
unshift(@array,3,4,5);      #@array=(3,4,5,6,7,8,9);
$x=shift(@array);      #$x=3,@array=(4,5,6,7,8,9);
```

和 pop 一样，如果返回一个空数组给 shift 的话，会得到 undef。

### 3.4.7 拼接函数——splice 函数

拼接函数可以向数组中间插入元素、删除子列表或替换子列表，并返回拼接后的列表。如果要将一个数组中的元数插入到另一个数组的中间，可以使用该函数。该函数有三种用法：

```
spliceARRAY,OFFSET,LENGTH,LIST
spliceARRAY,OFFSET,LENGTH
spliceARRAY,OFFSET
```

参数 OFFSET 是拼接前跳过的元素数目，LENGTH 是被替换的元素数，LIST 是将要拼接进来的数组。

下面是一些例子：

**程序清单 3-10：splice 函数的使用**

```
@members=("Time","Flies");
@initiates=("An","Arrow");
splice(@members,2,0,"Like",@initiates);
print"@members\n";
@removed=splice(@members,0,1,"Fruit");
splice(@members,-2,2,"A","Banana");
print"Removed:@removed\n";
print"@members\n";
```

打印结果如下：

```
TimeFliesLikeAnArrow
Removed:Time
FruitFliesLikeABanana
```

## 3.5　小结与练习

### 3.5.1　小结

本章介绍了数组和列表，它也是 Perl 5 的主要数据类型之一。

第 1 节首先介绍了数组和列表的定义，然后介绍了数组的建构及表示方法。

第 2 节介绍了对数组元素的赋值及存取操作。因为数组是顺序存储结构，所以可以通过整数下标来存取数组元素，也可以通过返回整数值的表达式作下标。

第 3 节介绍了另外两种对数组的重要操作，即数组的输出和改变数组中元素的方法。数组元素可以单独输出，也可以用 print 函数整体输出。同样，可以单独改变某个数组元素的值，也可以用循环方法对多个数组元素进行改变，还可以一次对多个数组进行操作。

本章最后一节介绍了一些常用的与数组有关的系统内置函数，这些函数可使对数组的操作变得更为简便。

下一章将重点介绍 Perl 的第 3 种基本数据结构——散列。

### 3.5.2　习题与思考

（1）给出下面程序的输出结果。

```
print(("BrightOrange","Burnt"),("CanaryYellow","Sunbeam"));
print(("BrightOrange","Burnt"),"Middle",("CanaryYellow","Sunbeam\n"));
print(1,2,7..10,14,15);
print"\n";
print("A","B","F"..."H","Y","Z");
```

（2）给出下面程序的输出结果。

```
@stuff=("one","two","three");
```

```
$stuff=@stuff;
print"Thelengthofthearrayis:$stuff\n;
print"@stuff";
@ary=(1,3,sort4,2);
print@ary;
```

（3）给出下面程序的输出结果。

```
@home=("couch","chair","table","stove");
print@home,"\n";
($potato,$lift,$tennis,$pipe)=@home;
print($potato,$lift,$tennis,$pipe,"\n");
($potato,$lift)=($lift,$potato);
print($potato,$lift,$tennis,$pipe);
```

（4）写一段程序，读入一连串字符串，存入列表中，然后把列表内容颠倒印出来。若你让程序从终端读取输入，最后可能要键入 end-of-file 字符代表输入结束。end-of-file 在 Linux 下以 Ctrl+D 代表，其他系统大部分是 Ctrl+z。

（5）读入一连串字符串，用随机数选取其中一行，把它打印出来。
提示：在程序最前面加入下面的语句：

```
srand
```

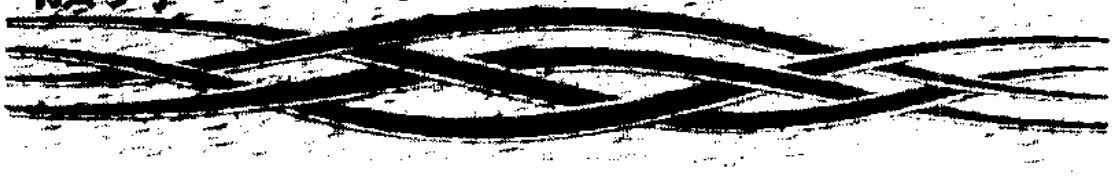
用来初始化随机数发生器，然后用

```
rand(@somearray)
```

得到介于 0 到@somearray 长度之间的随机数。

# 第4章

# 散列



散列简介

存取散列的元素

散列函数

散列应用举例

小结与练习



通过学习散列，读者将会看到，对 Perl 5 散列的操作比对 Perl 5 数组的操作更加有趣，因为它便于记忆，使用灵活。

本章要点：

- 散列与一般数组的区别和它们之间的相互转化。
- 访问散列中的元素。
- 散列函数的使用。

## 4.1 散列简介

### 4.1.1 什么是散列

散列也可直译为哈希（hash），过去的文档叫做关联数组（associative array），由于名字太长，所以改成现在这个名字。它与前面介绍过的数组非常类似，其中存放着许多标量变量，每个元素都可用索引值独立选取。和数组不同的是，散列不仅仅使用正整数当索引值，而是任何标量均可。这些标量称做 keys，与这些索引相关的值称为 values。

散列是一个计算机术语，表示有效地存放/读取稀疏数组的机制。稀疏数组就是索引很大而实际数值不多的数组。散列表用一种算法确定存放/读取数据值的索引，目的是对索引很大而实际数值不多的数组产生唯一索引，确定数值存放的位置。Perl 5 隐藏了所有算法细节，所以读者只需要知道，散列表用关键字查找相关数值存放的地址。

散列的元素没有特定的顺序，可以把它们想像成置于档案柜中的资料卡片。每张卡片的上半部是索引 keys，下半部是值 values。每次将新值放进散列时，会产生一张新卡片。其后若想更改其值，可以指定某个索引，让 Perl 选择到正确的卡片。事实上，Perl 会用特别的顺序存放 keys 和资料的序对，这样就不用在所有序对中搜寻所要的。

下面就是一个散列的例子：

```
%longday = ("Sun", "Sunday", "Mon", "Monday", "Tue", "Tuesday",
            "Wed", "Wednesday", "Thu", "Thursday", "Fri",
            "Friday", "Sat", "Saturday");
```

### 4.1.2 为什么要使用散列

读者可能会问，既然已经有数组了，为什么还要散列这种数据类型呢。答案就在于数组仍有许多操作上的不便，而散列恰好能够很好地解决这些问题。

在前面讲的数组变量中，可以通过下标访问其中的元素。例如，用下列语句访问数组 @array 的第三个元素：

```
$scalar = $array[2];
```

虽然数组很有用，但它们有一个显著缺陷，即很难记住哪个元素存储的是什么内容。假如写一个程序，要统计某文件中首字母大写的单词的出现次数，用数组来实现就比较困难，其程序代码如下：

**程序清单 4-1：用数组实现统计功能**

```
# This program finds the Capitalized words
while ($inputline = <STDIN>) {
    while ($inputline =~ /\b[A-Z]\S+/g) {
        $word = $&;
        $word =~ s/[;.,:-]$//;      #remove punctuation
        for ($count = 1; $count <= @wordlist; $count++) {
            $found = 0;
            if ($wordlist[$count-1] eq $word) {
              $found = 1;
              $wordcount[$count-1] += 1;
              last;
            }
        }
        if ($found == 0) {
          $oldlength = @wordlist;
          $wordlist[$oldlength] = $word;
          $wordcount[$oldlength] = 1;
        }
    }
}
print ("Capitalized words and number of occurrences:\n");
for ($count = 1; $count <= @wordlist; $count++) {
    print ("$wordlist[$count-1]: $wordcount[$count-1]\n");
}
```

运行结果如下：

```
Here is a line of Input.
This Input contains some Capitalized words.
^D
```

```
Capitalized words and number of occurrences:
Here: 1
Input: 2
This: 1
Capitalized: 1
```

这个程序每次从标准输入文件读入一行文字，然后循环匹配每行中首字母大写的单词，每找到一个循环一次，赋给简单变量$word。然后在删除标点后，查看该单词是否曾出现过，如果某个元素与$word 相等，@wordcount 中相应的元素就增加一个数。如果没有出现过，即@wordlist 中没有元素与$word 相等，最后给@wordlist 和@wordcount 增加一个新元素。

正如读者所看到的，使用数组元素将产生了一些问题。首先，@wordlist 中哪个元素对应着哪个单词并不明显；更糟的是，每读入一个新单词，程序必须检查整个列表才能知道该单词是否曾经出现过，当列表变得较大时，这是很耗费时间的。

这些问题产生的原因是数组元素通过数字下标访问。为了解决这类问题，Perl 定义了另一种数组，可以用任意简单变量值来访问其元素，这就是散列。

### 4.1.3 散列变量

散列变量的名称必须以百分号 % 开头，后面连接一个字母，之后可接更多字母、数字或下划线，这个规则和其他种类的变量一样。所以，$x、@x 和%x 这三个变量彼此之间没有任何关联。

可以用单个赋值语句创建散列，如：

```
%fruit = ("apples",17,"bananas",9,"oranges","none");
```

此语句创建的散列含有下面三个元素：
下标为 apples 的元素，值为 17。
下标为 bananas 的元素，值为 9。
下标为 oranges 的元素，值为 none。



用列表给散列赋值时，Perl 5 允许使用“=>”或“,”来分隔下标与值，用“=>”可读性更好些。

也就是说，上面的语句等效于：

```
%fruit = ("apples"=>17,"bananas"=>9,"oranges"=>"none");
```

总结一下，散列有下面三种表示方式：

1. 形式一

直接以数组的形式依次给出散列中的键和值：

```
%map = ('red',0xff0000,'green',0x00ff00,'blue',0x0000ff);
```

2. 形式二

以大括号的形式分别给出键和值：

```
%map = ();      # clear the hash first
$map{red} = 0xff0000;
$map{green} = 0x00ff00;
$map{blue}   = 0x0000ff;
```

3. 形式三

用“=>”符号依次给出键和值对，这种形式最为直观方便：

```
%map = (
    red    => 0xff0000,
    green => 0x00ff00,
    blue   => 0x0000ff,
);
```

下面是一个例子：

**程序清单 4-2：散列变量的应用**

```
%associativeArray = ("Jack A.", "Dec 2", "Joe B.",
 "June 2", "Jane C.", "Feb 13");
$associativeArray{"Jennifer S."} = "Mar 20";
print "Joe's birthday is: " . $associativeArray{"Joe B."} . "\n";
print "Jennifer's birthday is: " . $associativeArray{"Jennifer S."} . "\n";
```

程序执行后在屏幕上打印结果为：

```
Joe's birthday is: June 2
```

```
Jennifer's birthday is: Mar 20
```

### 4.1.4 全局散列变量

下面给出几个系统定义的全局散列变量，读者可以无需定义而使用它们，由于它们已包括特定的含义，所以用户不可以再使用相同的散列名。

%INC —— 散列，包含通过 do 或 require 包括的每个文件名项目（关键字即键是指定的文件名，值是文件的实际位置）。

%ENV —— 包含当前环境的散列 （在%ENV中设置数值可以改变子进程的环境）。

%SIG —— 设置各种信号的信号处理器的散列。

## 4.2 存取散列的元素

### 4.2.1 访问散列中的元素

一般来说，很少存取整个散列，而只是存取其中的元素。散列变量%var 的某个元素可用$var{$key}加以存取，其中$key 代表能返回标量值的语句。注意存取单独元素时，前面用的是“$”号而非“%”号。

散列的下标可以为任何简单标量值，访问单个元素时以“$”符号打头，下标用大括号括起来。例如：

```
$fruit{"bananas"}
$number{3.14159}
$integer{-7}
```

简单变量也可作为下标，如：

```
$fruit{$my_fruit}
```

存取不存在的元素，系统会返回 undef，这和数组一样。

下面的第三行的用法也会出错：

```
$days{'February 29th'}     # OK
$days{"February 29th"}     # Also OK
$days{ February 29th }     # WRONG
```

### 4.2.2 增加元素

创建一个散列元素最简单的方法是赋值，如语句：

```
$fruit{"bananas"} = 1;
```

把 1 赋给散列%fruit 下标为 bananas 的元素，如果该元素不存在，则被创建；如果散列%fruit 从未使用过，也被创建。

这一特性使得散列很容易用于计数。下面我们用散列方法改写上面提到过的统计首字母大写的单词个数的程序。虽然实现同样的功能，但下面的程序简化了许多。

**程序清单 4-3：用散列实现统计功能**

```
# This program finds the Capitalized words
while ($inputline = ) {
    while ($inputline =~ /\b[A-Z]\S+/g) {
        $word = $&;
        $word =~ s/[:.,:-]$//;    # remove punctuation
        $wordlist{$word} += 1;
    }
}
print ("Capitalized words and number of occurrences:\n");
foreach $capword (keys(%wordlist)) {
    print ("$capword: $wordlist{$capword}\n");
}
```

运行结果如下：

```
Here is a line of Input.
This Input contains some Capitalized words.
^D
Capitalized words and number of occurrences:
This: 1
Input: 2
Here: 1
Capitalized: 1
```

可以看到，使用了散列，这个程序简单多了。本程序用散列%wordlist 跟踪首字母大写的

单词，下标就用单词本身，元素值为该单词出现的次数。其中使用了内嵌函数 keys()。这个函数返回散列的下标列表，foreach 语句就用此列表循环。

散列总是随机存储的，因此当用 keys()访问其所有元素时，不保证元素以任何顺序出现，特别值得一提的是，它们不会以被创建的顺序出现。

要想控制散列元素出现的次序，可以用 sort()函数对 keys()返回值进行排列，如：

```
foreach $capword (sort keys(%wordlist)){
    print ("$capword: $wordlist{$capword}\n");
}
```

### 4.2.3 数组与散列的相互转换

与列表一样，也可以通过数组变量创建散列，当然，其元素数目应该为偶数，如：

```
@fruit = ("apples",3,"bananas",5,"oranges",2);
%fruit = @fruit;      # 相当于 %fruit = ("apples"=>3, "bananas"=>5,"oranges"=>2);
```

反之，也可以把散列赋给数组变量，如：

```
%fruit = ("apples"=>3, "bananas"=>5,"oranges"=>2);
@fruit = %fruit;      # 此时 @fruit = ("apples",3,"bananas",5,"oranges",2);
```

注意，由于散列中元素次序未定义，那么数组变量@fruit 可能为("bananas",5, "apples", 3,"oranges",2)或其他。

散列变量之间可以直接赋值，如：

```
%fruit2 = %fruit1;
```

还可以把数组变量同时赋给一些简单变量和一个散列变量，如：

```
$var1, $var2, %hashname) = @list;
```

上面的语句把@list 中的第一个元素赋给$var1，第二个赋给$var2，其余的赋给散列%hashname。

下面是一个将散列赋给数组的例子：

**程序清单 4-4：将散列赋值给数组**

```
%father = (   'Cain '        => 'Adam ',
              'Abel '        => 'Adam ',
              'Seth '        => 'Adam ',
              'Enoch '       => 'Cain ',
              'Irad '        => 'Enoch ',
              'Mehujael '    => 'Irad ',
              'Methusael '   => 'Mehujael ',
              'Lamech '      => 'Methusael ',
              'Jabal '       => 'Lamech ',
              'Jubal '       => 'Lamech ',
              'Tubalcain '   => 'Lamech ',
'Enos '       => 'Seth ' );
@father = %father;
print @father;
```

在屏幕上打印结果为：

```
Abel Adam Enos Seth Jabal Lamech Cain Adam Jubal Lamech Enoch Cain Mehujael Irad
 Tubalcain Lamech Lamech Methusael Seth Adam Irad Enoch Methusael Mehujael
```

## 4.3 散列函数

表 4-1 为常用的散列函数列表，读者在使用时可以随时查询。

表 4-1 常用的散列函数

| 函数名 | 调用语法 | 返回值 | 说明 |
|---|---|---|---|
| keys | @list = keys (%assoc_array); | 返回关联数组无序的下标列表 | |
| values | @list = values (%assoc_array); | 返回关联数组无序的值列表 | |
| each | @pair = each (%assoc_array); | 返回两个元素的列表——键值（即下标和相应的值）也是无序的 | 当关联数组已空，则返回空列表 |

续表

| 函数名 | 调用语法 | 返回值 | 说明 |
| --- | --- | --- | --- |
| delete | element = delete (assoc_array_item); | 返回值为删除的元素 | 删除关联数组中的元素 |
| exists | result = exists (element); | 若存在，返回非零值，否则返回零值 | 在 Perl 5 中定义，判断关联数组中是否存在某个元素 |

### 4.3.1 keys 函数

keys(%hashname)函数将返回一个列表，内容是散列变量%hashname 里的所有 keys。换句话说，它能返回数组环境下%hashname 的单数位置元素。若散列内不含任何元素，该函数返回空列表。

请看下面的例子：

```
%fruit = ("apples"=>4, "pears"=>5,"peachs"=>9);
@list = keys(%fruit);    # @list 的值是 （4, 9）或（9,4）
```

这里，@list 被赋给 apples、peachs、pears 构成的列表，再次提请注意，此列表没有次序，若想按字母顺序排列，可使用 sort()函数：

```
@fruitIndexes = sort keys(%fruits);    # 此时 @fruitIndexes = （"apples","peachs", "pears"）
```

和其他的内置函数一样，参数前后的小括号可以省略："keys %fruit" 就等于 "keys (%fruit)"。

我们可以把单独的散列元素插入双引号字符串中，但对于整个散列变量就不行了。

在标量环境下，keys 函数会返回散列中的序对数。用法如下：

```
%fruit = ("apples"=>17,"bananas"=>9);
$x = keys (%fruit);    # $x = 2;
```

下面给出一个完整的例子：

**程序清单 4-5：keys 函数的应用**

```
%food_color = (   Apple   => "red",
                  Banana => "yellow",
                  Lemon   => "yellow",
                  Carrot => "orange"
```

```
                );
$food_color{Raspberry} = "pink";
print "Known foods:\n";
foreach $food (keys %food_color) {
    print "$food\n";
}
```

打印结果为：

```
Known foods:
Apple
Carrot
Lemon
Banana
Raspberry
```



由于散列的存储是不受控制的，所以不同的机器上运行输出的次序可能不同。

### 4.3.2 values 函数

values(%hashname)函数和 keys(%hashname)一样，能用列表的形式返回%hashname 里所有 key-value 序对的 value 值。请看下面的例子：

```
%fruit = ("apples"=>17,"bananas"=>9);
@valueList = values (%fruit);     #@valueList = (17, 9);
```

与 keys 函数一样，values 参数前后的小括号也可以省略："values %fruit" 就等于 "values (%fruit)"。

下面是一个针对系统环境的应用例子：

**程序清单 4-6：values 函数的应用**

```
@keys   = keys   %ENV;
@values = values %ENV;
while (@keys) {
    print pop(@keys), '=', pop(@values), "\n";
}
foreach $key (sort keys %ENV) {
    print $key, '=', $ENV{$key}, "\n";
```

}

屏幕上打印出的是正在运行的环境，每台具体的计算机会有所不同。



values 函数返回的清单与 keys 函数返回的清单的顺序相同，因为它们参照的是同一种内部存储顺序。

### 4.3.3 each 函数

each 函数不像 keys 和 values 函数那样分别返回散列中所有索引/数值清单，它每次返回一个双元素的列表，其第一个元素为索引，第二个元素为相应的值。

keys 和 values 函数返回的是散列中所有索引/数值的完整清单，如果散列有几百万个项目，则这个清单会占用大量的计算机内存。而 each 函数则可以一次一个索引/数值对地对散列进行遍历，而且一次取得索引和数值，使用起来更方便。下面是一个例子：

**程序清单 4-7：each 函数的应用**

```
%fruit = ("apples"=>3,"bananas"=>9, "oranges"=>2, "pears"=>4, "peachs"=>6);
while (($fruit,$price) = each %fruit){
    print "The price of the $fruit is $price. \n";
}
```

执行结果是在屏幕上打印出：

```
The price of the apples is 3.
The price of the bananas is 9.
The price of the oranges is 2.
The price of the pears is 4.
The price of the peachs is 6.
```

当然，以上的输出顺序在实际执行时可能会有所不同。



千万不要在 each()循环中添加或删除元素，否则会产生不可预料的后果。

### 4.3.4 delete 函数

要删除散列元素，可利用 Perl 提供的 delete 函数。它以散列元素的索引值做为参数，并删除该序对。请看下面的例子：

```
%fruit = ("apples"=>3,"bananas"=>9, "oranges"=>2, "pears"=>4, "peachs"=>6);
```

```
delete $fruit{"bananas"};
```

执行后，%fruit 散列中只剩下 4 个序对：

```
%fruit = ("apples"=>3, "oranges"=>2, "pears"=>4, "peachs"=>6);
```

下面是一个完整的例子：

**程序清单 4-8：delete 函数的应用**

```
# %food_color as per Introduction
sub print_foods {
    my @foods = keys %food_color;
    my $food;
    print "Keys: @foods\n";
    print "Values: ";
    foreach $food (@foods) {
        my $color = $food_color{$food};
        if (defined $color) {
            print "$color ";
        }
else {
            print "(undef) ";
        }
    }
    print "\n";
}
%food_color = (  Apple  => "red",
                 Banana => "yellow",
                 Lemon  => "yellow",
                 Carrot => "orange"
                );
print "Initially:\n";
print_foods();
print "\nWith Banana undef\n";
undef $food_color{"Banana"};
print_foods();
print "\nWith Banana deleted\n";
delete $food_color{"Banana"};
```

```
print_foods();
```

在这个例子中，用到了一个 print_foods 函数，实际上它与我们在其他语言中使用的函数差不多，都是为实现特定的功能而设计的，这里它的功能是打印出散列中的键和值。该程序的执行结果在屏幕上打印出：

```
Initially:
Keys: Apple Carrot Lemon Banana
Values: red orange yellow yellow

With Banana undef
Keys: Apple Carrot Lemon Banana
Values: red orange yellow (undef)

With Banana deleted
Keys: Apple Carrot Lemon
Values: red orange yellow
```



一定要使用 delete 函数来删除散列的元素，这是唯一的方法。一定不要对散列使用内嵌函数 push、pop、shift 及 splice，因为其元素位置是随机的。

## 4.4 散列应用举例

散列是 Perl 中的一类重要的数据结构，在其他高级语言中很少有散列，因此散列也是 Perl 的特点之一。散列有着广泛的应用，下面就给出两个例子。

### 4.4.1 用散列实现多维数组

Perl 中没有定义多维数组，但用户可以自己实现多维数组，使用散列就是一种很好的方法。

**程序清单 4-9：用散列实现多维数组**

```
%wife = ();
%kids_of_wife = ();
$wife{"Jacob"} = ("Leah", "Rachel", "Bilhah", "Zilpah");
$wife{"Jacob"}[0] = "Leah";
```

```
$wife{"Jacob"}[1] = "Rachel";
$wife{"Jacob"}[2] = "Bilhah";
$wife{"Jacob"}[3] = "Zilpah";
print "The wifes of Jacob are : \n";
for($i = 0;$i < 4;$i++) {
    print $wife{"Jacob"}[$i]," ";
}
$kids_of_wife{"Jacob"} = {
    "Leah"   => ["Reuben", "Simeon", "Levi","Judah", "Issachar", "Zebulun"],
    "Rachel" => ["Joseph", "Benjamin"],
    "Bilhah" => ["Dan", "Naphtali"],
    "Zilpah" => ["Gad", "Asher"],
};
$kids_of_wife{"Jacob"}{"Leah"}[0]    = "Reuben";
$kids_of_wife{"Jacob"}{"Leah"}[1]    = "Simeon";
$kids_of_wife{"Jacob"}{"Leah"}[2]    = "Levi";
$kids_of_wife{"Jacob"}{"Leah"}[3]    = "Judah";
$kids_of_wife{"Jacob"}{"Leah"}[4]    = "Issachar";
$kids_of_wife{"Jacob"}{"Leah"}[5]    = "Zebulun";
$kids_of_wife{"Jacob"}{"Rachel"}[0] = "Joseph";
$kids_of_wife{"Jacob"}{"Rachel"}[1] = "Benjamin";
$kids_of_wife{"Jacob"}{"Bilhah"}[0] = "Dan";
$kids_of_wife{"Jacob"}{"Bilhah"}[1] = "Naphtali";
$kids_of_wife{"Jacob"}{"Zilpah"}[0] = "Gad";
$kids_of_wife{"Jacob"}{"Zilpah"}[1] = "Asher";

for($i = 0;$i < 4;$i++){        // 输出
    print "\nThe kids of wife ",$wife{"Jacob"}[$i]," are :\n";
    for($j=0;$j<6;$j++){
        print $kids_of_wife{"Jacob"}{$wife{"Jacob"}[$i]}[$j]," ";
    }
}
```

程序执行结果为在屏幕上打印出：

```
The wifes of Jacob are :
Leah Rachel Bilhah Zilpah
The kids of wife Leah are :
```

```
Reuben Simeon Levi Judah Issachar Zebulun
The kids of wife Rachel are :
Joseph Benjamin
The kids of wife Bilhah are :
Dan Naphtali
The kids of wife Zilpah are :
Gad Asher
```

该程序中有两个散列，%wife 和 %kids_of_wife ，前者是某人（“Jacob”）的妻子的散列，后者是各位妻子的孩子的散列。然后将妻子的散列用来表示一维数组，每个具体的元素表示一位妻子；将妻子的孩子的散列用二维数组表示，其中一维用来表示妻子，另外一维用来表示孩子，每一个具体的元素表示某位妻子的某个孩子。程序中一共有两处输出：一个是输出妻子散列表示的数组的每个元素——每个妻子；另一个是输出妻子的孩子散列表示的二维数组的每个元素——每个孩子。

### 4.4.2 用散列实现结构

用散列还可以模拟在其他高级语言中常见的多种数据结构，如链表、结构和树等，本节讲述如何用它来实现结构。

许多编程语言可以定义结构，即一组数据的集合。结构中的每个元素有其自己的名字，并通过该名字来访问。Perl 虽然不直接提供结构这种数据结构，但可以用散列来模拟。例如模拟 C 语言中如下的结构：

```
struct {
        int field1;
        float field2;
        char field3;
} mystruct;
```

我们要做的是定义一个含有三个元素的散列，下标分别为 field1、field2、field3，如：

```
%mystruct = ("field1" , "" , "field2" , "" , "field3" , "" ,);
```

像上面 C 语言的定义一样，这个散列%mystruct 有三个元素，下标分别为 field1、field2 和 field3，各元素初始值均为空字符串。对各元素的访问和赋值可通过指定下标来进行，如：

```
$mystruct{"field1"} = 23;
$mystruct{"field2"} = 5.67;
$mystruct{"field1"} = 'a';
```

# 4.5 小结与练习

## 4.5.1 小结

本章介绍了散列，它也是 Perl 5 的主要数据类型之一。

第 1 节首先介绍了散列的定义，以及为什么要使用散列。然后介绍了散列变量的三种表示形式，其中推荐使用第三种形式。本节还介绍了几个常用的散列变量。

第 2 节介绍了对散列中的元素的存取操作。怎么向散列中增加元素，以及如何在数组与散列之间相互转换，本节都做了介绍。

第 3 节介绍了一些常用的与散列有关的系统内置函数。keys 函数可以取出散列的键，values 函数可以取出散列的值，each 函数可以同时取出散列的键和值；delete 函数可以方便地删除散列中的元素。这些函数将使对散列的操作十分简便。

本章最后一节给出了两个例子来说明散列的应用。一个是用散列实现多维数组，另一个是用散列实现结构。这些例子说明了散列在 Perl 中的独特作用。

散列是 Perl 的重要特点之一，也是它的三个重要的基本数据结构之一。学习了本章，读者应该掌握 Perl 中散列的使用。至此，Perl 的三种基本数据结构——标量、数组和散列就介绍完了。下一章将介绍 Perl 中的操作符。

## 4.5.2 习题与思考

（1）写出下面程序的执行结果。

```
%associativeArray = ("Jack A.", "Dec 2", "Joe B.", "June 2", "Jane C.", "Feb 13");
$associativeArray{"Jennifer S."} = "Mar 20";
print "Joe's birthday is: " . $associativeArray{"Joe B."} . "\n";
delete $associativeArray{"Joe B."};
print "Jennifer's birthday is: " . $associativeArray{"Jennifer S."}. "\n";
```

（2）写一个程序，当输入表 4-2 中左边的颜色值时，输出相应的事物名。

表 4-2 事物与颜色对应表

| 输　入 | 输　出 |
| --- | --- |
| red | Apple |

续表

| 输　入 | 输　出 |
| --- | --- |
| green | Leaves |
| blue | Ocean |
| yellow | Banana |
| orange | Carrot |

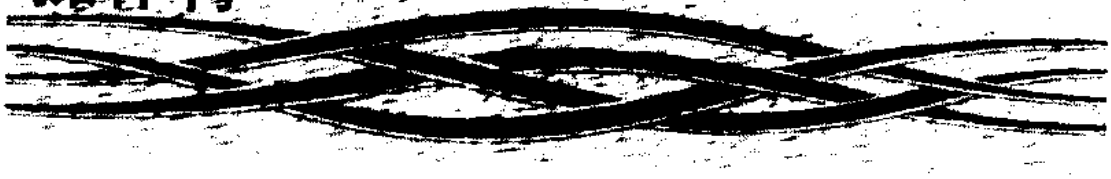
（3）写出下面程序的输出结果：

```
%food_color = (  Apple   => "red",
                 Banana => "yellow",
                 Lemon   => "yellow",
                 Carrot => "orange"
                );
$food_color{Raspberry} = "pink";
print "Known foods:\n";
foreach $food (keys %food_color) {
    print "$food\n";
}
```

（4）写一个程序，读入一系列单词，每个单词占一行，然后统计每个单词出现的次数。

# 第5章

# 操作符



数字操作符
字符串操作符
逻辑操作符和位操作符
其他操作符
操作符的优先级和结合性
小结与练习



介绍完几种基本的数据类型之后，我们再来介绍一下操作符。

操作符能从一个或一个以上的运算元中产生出新的值。操作符的运算元可能是数字，也可能是字符串，甚至是两者的组合。若程序设计人员在应该给出数字的地方使用了字符串，或在应该给出字符串的地方使用了数字，Perl 会自动地根据合理的规则加以转换。

数字操作符、字符串操作符、逻辑操作符和位操作符是几类常用的操作符，另外还有一些比较特殊的操作符。这些操作符的次序和结合性也是非常重要的。

本章要点：

- 几类常用的操作符及其用法。
- 几个特殊的操作符。
- 各种操作符的优先级和结合性。

# 5.1 数字操作符

## 5.1.1 基本数字操作符

表 5-1 列出了 Perl 5 的几种基本数字操作符，它们都是大家所熟悉的。

表 5-1 几种基本数字操作符

| 操作符 | 例子 | 说明 |
|---|---|---|
| + | $a = 5 + 8 | 将操作符左边的表达式加上操作符右边的表达式，$a =13 |
| - | $a = 15 - 8 | 将操作符左边的表达式减去操作符右边的表达式，$a =7 |
| * | $a = 5 * 8 | 将操作符左边的表达式乘以操作符右边的表达式，$a =40 |
| / | $a = 2 / 8 | 将操作符左边的表达式除以操作符右边的表达式，$a =0.25 |
| ** | $a = 5 ** 2 | 操作符左边的表达式为底数以操作符右边表达式为指数的幂，$a =25 |
| % | $a = 15 % 8 | 将操作符左边的表达式用操作符右边的表达式求模，$a =7 |
| - | $a = - $b | 将操作符右边的表达式符号取反 |

下面是使用上述操作符的一个例子：

```
printf("%.51f\n", 5.1 - 2.4);
print(5.1 - 2.4, "\n");
```

该程序的执行结果是在屏幕上输出：

```
2.69999999999999973354647408996243029832839965820312 5
2.7
```

第一个结果之所以这么长，是由函数 printf 的格式限定引起的，我们将在第 12 章中具体讲解格式化输出，有兴趣的读者可以查看一下。

虽然这些基本算术操作符比较简单，但读者在使用它们时，仍需要注意以下几点：

- 乘幂的基数不能为负，如 (-5) ** 2.5 就是错误的。
- 乘幂结果不能超出计算机表示的限制，如 10 ** 999999 就会溢出。
- 取余的操作数如不是整数，四舍五入成整数后运算；操作符右侧不能为零。
- 单目负操作符可用于变量： 如- $y 等效于 $y * (-1)。

### 5.1.2 数字比较操作符

数字比较操作符也是大家比较熟悉的，表 5-2 列出了这些操作符。

表 5-2　　数字比较操作符

| 操 作 符 | 描　述 |
|---|---|
| < | 小于 |
| > | 大于 |
| == | 等于 |
| <= | 小于等于 |
| >= | 大于等于 |
| != | 不等于 |
| <=> | 比较，返回 1， 0， 或者 -1 |



操作符<=>的结果为：

0，当两个值相等时；

1，当第一个值大时；

-1，当第二个值大时。

下面是一个使用比较操作符的例子：

**程序清单 5-1：比较操作符的使用**

```
$lowVar =  8;
$midVar = 10;
$hiVar  = 12;
```

```
print($lowVar <=> $midVar, "\n");
print($midVar <=> $midVar, "\n");
print($hiVar  <=> $midVar, "\n");
```

程序执行的输出为：

```
-1
0
1
```

### 5.1.3 赋值操作符

表 5-3 给出了常用的赋值操作符。

表 5-3　　赋值操作符

| 操 作 符 | 表 达 式 | 等效表达式 |
|---|---|---|
| = | $a = 1; | |
| += | $a += 1; | $a = $a + 1; |
| -= | $a -= 1; | $a = $a - 1; |
| *= | $a *= 2; | $a = $a * 2; |
| /= | $a /= 2; | $a = $a / 2; |
| %= | $a %= 2; | $a = $a % 2; |
| **= | $a **= 2; | $a = $a ** 2; |
| &= | $a &= 2; | $a = $a & 2; |
| \|= | $a \|= 2; | $a = $a \| 2; |
| ^= | $a ^= 2; | $a = $a ^ 2; |

下面是应用上述赋值操作符的几个例子：

**程序清单 5-2：赋值操作符的应用举例**

```
$b = 5;
$a = $b;
print $a."\n";
$a = $a * 3;
$b **= 3;
print $a,$b."\n";
$line = "a line.";
```

```
$line .= "\n";
print $line."\n";
$fill = "OK";
$fill x= 8;
print $fill."\n";
$val = "0";
$val ll= "2";
print $val."\n";
$a = $b = $c = 0;
print $a,$b,$c."\n";
$temp = 2;
($temp -= 32) *= 5/9;
print $temp."\n";
```

在屏幕上输出结果为：

```
5
15125
a line.

OKOKOKOKOKOKOKOK
2
000
-16.6666666666667
```

值得一提的是，赋值操作符（=）可在一个赋值语句中多次出现，上面的例子中已经出现了，又如：

```
$value1 = $value2 = "a string";
```

等价于：

```
$value1 = "a string";
$value2 = "a string";
```

“=”还可以作为子表达式，例如：

```
($a = $b) += 3;
```

等价于：

```
$a = $b;
$a += 3;
```

但建议不要使用这种方式，否则会使程序很难读懂。

### 5.1.4 自增自减操作符

自增操作符为：++；自减操作符为：--，它们的使用与 C 语言中基本相同，下面是几个例子：

```
$x = 12;
$x++;    # $x 现在是 13
$x--;    # $x 现在是 12
```

需要说明的是，操作符“++”和“--”放在变量的左边和右边，其效果是不同的。当它放在左边时，返回的结果是原值增加或减少 1 之后的值；而放在右边时，返回的值仍是原来的值。在执行完含有自增自减操作符的语句之后，不管操作符是放在左边还是右边，该操作符所针对的变量都会增加或减少 1，这一点是相同的。下面用一些例子来更进一步地说明这种区别：

```
$a = 10;
$b = ++$a;       # 现在 $b = 11，  $a = 11
$a = 10;
$c = $a++;       # 现在 $c = 10，  $a = 11
$a = 10;
$d = --$a;       # 现在 $d = 9，   $a = 9
$a = 10;
$e = $a--;       # 现在 $e = 10，  $a = 9
```

注意：不要在变量两边都使用此种操作符。

```
++$var--         # 这样写是错误的
```

不要在变量自增自减后在同一表达式中再次使用。

```
$var2  = $var1 + ++$var1;     # 这样写是错误的
```

在 Perl 中“++”可用于字符串，但当结尾字符为‘z’、‘Z’或‘9’时进位，如：

```
$stringvar = "abc";
$stringvar++;      # $stringvar 现在是 "abd"
$stringvar = "aBC";
$stringvar++;      # $stringvar 现在是 "aBD"
$stringvar = "abz";
$stringvar++;      # $stringvar 现在是 "aca"
$stringvar = "AGZZZ";
$stringvar++;      # $stringvar 现在是 "AHAAA"
$stringvar = "ab4";
$stringvar++;      # $stringvar 现在是 "ab5"
$stringvar = "bc999";
$stringvar++;      # $stringvar 现在是 "bd000"
```

对字符串不要使用--，Perl 会先将字符串转换为数字再进行自减：

```
$stringvar = "abc";
$stringvar--;      #   $stringvar = -1
```



++作用于字符串是 Perl 的一个特点，但 --不要用于字符串。

如果字符串中含有非字母且非数字的字符，或数字位于字母中，则经过++运算，前值转换为数字零，因此结果为 1，如：

```
$stringvar = "ab*c";
$stringvar++;      #   $stringvar = 1
$stringvar = "ab5c";
$stringvar++;      # $stringvar = 1
```

在数组下标中运用自增自减操作符，可使程序变得十分简便，类似于 C 语言中的循环变量的自增和自减：

```
$var[$a++] += $value;      # $a增加1
$var[$a++] = $var[$a++] + $value;      # $a 增加2
```

# 5.2 字符串操作符

## 5.2.1 字符串比较操作符

前面曾经提到过，Perl 会自动对字符串和数字进行转换。但是同样的数，把它当字符串或数字处理其结果是不同的。例如，数字 7 当然比数字 30 小，但如果把它们当字符串比较，因为字符 3 的 ASCII 编码比字符 7 的要小，“30”会在“7”之前，所以答案是小于。正因为如此，指定数据类型究竟是数字或字符串仍然是必要的，而且它们所用的操作符是不同的。

下面是字符串比较操作符的列表，读者可以把它们与数字比较操作符进行一下比较。

表 5-4 字符串比较操作符

| 操 作 符 | 描 述 |
|---|---|
| lt | 小于 |
| gt | 大于 |
| eq | 等于 |
| le | 小于等于 |
| ge | 大于等于 |
| ne | 不等于 |
| cmp | 比较，返回 1、0、or -1 |

下面是一个使用字符串比较操作符的例子：

**程序清单 5-3：字符串比较操作符的使用**

```
$lowVar = "AAA";
$midVar = "BBB";
$hiVar   = "ccC";
print($lowVar cmp $midVar, "\n");
print($midVar cmp $midVar, "\n");
print($hiVar   cmp $midVar, "\n");
```

程序输出结果为：

```
-1
0
1
```

## 5.2.2　字符串联接和重复操作符

1. 字符串联接操作符

字符串联接操作符为一个句点号（.），下面是一些例子：

```
"hello" . "world"      # same as "helloworld"
'hello world' . "\n"      # same as "hello world\n"
"fred" . " " . "barney"      # same as "fred barney
$newstring = "fore" . "head";      # $newstring 变为"forehead"
```

与赋值操作符（+=）等类似，Perl 中也有联接且赋值操作符（.=），下面就是使用该操作符的例子：

```
$a = "wel";
$a .= "fare";      #$a 现在是 "welfare"
```

下面的程序将清楚地表示出联结操作符的作用：

**程序清单 5-4：联结操作符的使用**

```
$firstname = "Fred";
$lastname = "Flintstone";
$almost = "Fred" . "Flintstone";
print $almost."\n";
$fullname = "$firstname $lastname";
print $fullname."\n";
$almost = "$firstname"."$lastname";
print $almost."\n";
```

程序执行后输出为：

```
FredFlintstone
Fred Flintstone
FredFlintstone
```

2. 字符串重复操作符

字符串重复操作符为："×"。注意，它不同于乘法操作符"*"，读者千万别混淆了。下面是一个例子：

```
$newstring = "a" x 4;    # $newstring 变为"aaaa"
$newstring = "good" x 3;    # $newstring 变为"goodgoodgood"
$newstring = "hey" x (3+2);    # $newstring 变为"heyheyheyheyhey "
(3+2) x 4;    # 5 x 4，即"5" x 4，结果是"5555"
(3+2)*4;    # 即5 * 4，结果是 20
```

注意，在最后一个例子中，小括号强迫其内部的 3+2 优先运算，所以答案是 5。由于×操作符需要左侧有一个字符串当作运算元，所以 Perl 会把 3+2 的结果——数字 5 换成字符串"5"，因此最后的答案就会是"5555"。如果我们把左右两个运算元颠倒过来，答案会是"44444"。由此例子可知，字符串的复制运算不满足交换律。

若有必要，×操作符会把左右两边的"数字"无条件舍去成整数。于是，×右边若是 4.8，它会被当成 4 来运算；×左边的数字运算结果若小于 0，它会被当成空字符串来处理。

下面的程序将有助于读者更好地理解字符串重复操作符：

**程序清单 5-5：字符串重复操作符的使用**

```
print '-' x 60;    # print row of dashes
print "\t" x ($tab/8), ' ' x ($tab%8);    # tab over
$ones = (1) x 30;    # a list of 80 1's
print $ones."\n";
@keys = qw(perls before swine after hard);
for($i = 0;$i<5;$i++){
    $key = $keys[$i];
    $hash{$key} = ("p") x $i;
}
foreach $word (keys %hash) {
print "The value of the key: $word is $hash{$word}\n";
}
```

该程序的执行结果为：

```
------------------------------------------------------------
1111111111111111111111111111111
The value of the key: swine is pp
```

```
The value of the key: after is ppp
The value of the key: before is p
The value of the key: perls is
The value of the key: hard is pppp
```

# 5.3　逻辑操作符和位操作符

## 5.3.1　逻辑操作符

逻辑操作符有以下 4 种：
逻辑与：$a && $b 。
逻辑或：$a || $b 。
逻辑非：！$a。
逻辑异或：$a xor $b。
下面是运用逻辑操作符的一个例子：

**程序清单 5-6：逻辑操作符的使用**

```
$firstVar = 10;
$secondVar = 10;
if ($firstVar == 10 && $secondVar == 9) {
    print("Yes &&!\n");
}
else{
    print "No &&!\n";
}
if ($firstVar == 10 || $secondVar == 10) {
    print("Yes ||!\n");
}
else{
    print "No ||!\n";
}
$firstVar = 10;
$secondVar = !$firstVar;
if ($secondVar == 0) {
   print("zero\n");
}
```

程序执行结果为：

```
No &&!
Yes ||!
zero
```

### 5.3.2 位操作符

位操作符有以下几种：
位与："&" 将两个数按位做与运算。
位或："|" 将两个数按位做或运算。
位非："~" 将某个数按位做非运算。
位异或："^" 将两个数按位做异或运算。
左移："$x << 1" 将数$x 向左移一位。
右移："$x >> 2" 将数$x 向右移两位。



不要将&用于负整数，因为 Perl 将会把它们转化为无符号数。

下面给出了一个例子，读者从它的输出结果中可以体会出位操作符的用法：

**程序清单 5-7：位操作符的使用**

```
print "4 | 3 = ",(4 | 3),"\n";
print "\"1.5\" & \"2.6\" = ",("1.5" & "2.6"),"\n";
print "\"1.5\" & 2.6 = ","1.5" & 2.6,"\n";
print "1.4 & 2.5 = ",1.4 & 2.5,"\n";
print "12 & 34 = ",12 & 34,"\n";
print "~12 = ",~12,"\n";
print "12^23 = ",12^23,"\n";
print "1 << 4 = ",1 << 4,"\n";
print "32 >> 4 = ",32 >> 4,"\n";
```

程序执行结果为：

```
4 | 3 = 7
"1.5" & "2.6" = 0.4
"1.5" & 2.6 = 0
1.4 & 2.5 = 0
```

```
12 & 34 = 0
~12 = 4294967283
12^23 = 27
1 << 4 = 16
32 >> 4 = 2
```

## 5.4 其他操作符

### 5.4.1 逗号操作符

逗号操作符没有什么特殊之处，甚至不能算是严格意义上的操作符。使用此操作符的唯一理由是提高程序的可读性，将关系密切的两个表达式结合在一起。逗号前面的表达式先进行运算，然后是它后面的表达式。如：

```
$var1 += 1, $var2 = $var1;
```

等价于

```
$var1 += 1;
$var2 = $var1;
```

又如：

```
$val = 26;
$result = (++$val, $val + 5);        #   $result = 32
```

注意，如果此处没有括号则意义不同：

```
$val = 26;
$result = ++$val, $val + 5;
```

等价于

```
$val = 26;
$result = ++$val;      #$result = 27
$val + 5;
```

### 5.4.2 条件操作符

与 C 语言类似，条件操作符“？：”的调用语法为：

```
条件?值1:值2
```

当条件为真时取值 1，为假时取值 2，如：

```
$var =1 0
$result = $var == 0 ? 14 : 7;  #   $result = 7
```

还可以像下面这样：

```
$result = 43 + ($divisor == 0 ? 0 : $dividend / $divisor);
```

Perl 5中，还可以在赋值式左边使用条件操作符来选择被赋值的变量，如：

```
$condvar = 43;
$condvar == 43 ? $var1 : $var2 = 14;        # $var1 = 14
$condvar = 34;
$condvar == 43 ? $var1 = 14 : $var2 = 14;       # $var2 = 14
```

下面是另外一个例子：

**程序清单 5-8：条件操作符的使用**

```
$firstVar = 1;
$secondVar = 1;
$thirdVar = 1;
($thirdVar == 0 ? $firstVar : $secondVar) = 10;
print "$firstVar\n";
print "$secondVar\n";
print "$thirdVar\n";
```

程序的输出为：

```
1
10
1
```

因为第四句中条件操作符选中的是$secondVar，所以输出的结果为$secondVar = 10。
下面是一个求某年是否是闰年的例子，其中使用了多个条件操作符：

**程序清单 5-9：多个条件操作符的使用**

```
$year = 2000;
$leapyear =
    $year % 4 == 0
        ? $year % 100 == 0
          ? $year % 400 == 0
            ? 1
            : 0
          : 1
        : 0;
if($leapyear == 1){
    print "Yes.\n";
}
else{
    print "No.\n";
}
$year = 2001;
$leapyear =
    $year %   4 ? 0 :
    $year % 100 ? 1 :
    $year % 400 ? 0 : 1;
if($leapyear == 1){
    print "Yes.\n";
}
else{
    print "No.\n";
}
```

程序输出的结果为：

```
Yes.
No.
```

注意，像下面这样使用条件操作符往往会得到错误的结果：

```
$a = 3;
```

```
($a % 2) ? $a += 10 : $a += 2;
print $a,"\n";
```

程序执行的结果是屏幕打印出：

```
15
```

这显然与所要得到的结果不同，原因是$a 加了两次，要想得到正确的结果可以加上括号，如：

```
$a = 3;
($a % 2) ? ($a += 10) : ($a += 2);
print $a,"\n";
```

程序执行的结果是：

```
13
```

为了避免错误，最好不要采用这种写法，而用比较清晰的不会引起混淆的写法。

## 5.5 操作符的优先级和结合性

程序中，常常会有两个操作符配以三个运算元共同计算的情形。为了避免模棱两可，就要靠操作符优先级的定义决定该行程序的语意究竟如何进行计算。例如，在语句“2+3*4”中，到底是加法先算，还是乘法先算呢？如果加法先算，此语句等价于“5*4”，答案是 20。不过大家都知道，数学运算规则是“先乘除，后加减”，所以正确答案应该是 2+12＝14 才对。以 Perl 的说法，这是因为乘法的优先级比加法要高。

虽然所有操作符都有“预定”的优先级顺序，不过我们可以用小括号强迫某语句必须要先计算，这和数学上的用法一样。因此，如果真的一定要让加法先做，把加法括起来就可以了。比如上式就可以写成“(2+3)*4”，答案是 20。

乘法和加法因为常用，所以优先性不成问题。但其他操作符，比如字符串联结以及指数操作符等，就没有这么简单了。遇到这种情形，最好还是参考一下对优先级的规定，不要凭直觉想当然。表 5-5 列出了各种操作符的优先级次序和它们的结合性，优先级高的放在表的前面。如果在 Perl 程序中看到了 C 语言中也有的操作符，那么其优先级和 C 语言中的定义是一样的。

表 5-5　　操作符的优先级（由高到底）和结合性

| 操作符 | 描　述 | 结合性 |
|---|---|---|
| ()"`` | 括号，函数，引号 | 左 |
| -> | 删除引用（访问）操作符 | 左 |
| ++, -- | 自增，自减 | 无 |
| -, ~, ! | 单目 | 右 |
| ** | 指数 | 右 |
| =~, !~ | 约束，非约束 | 左 |
| *, /, %, x | 乘，除，取余，重复 | 左 |
| +, -, . | 加，减，联接 | 左 |
| <<, >> | 移位 | 左 |
| <, <=, >, >=, lt, le, gt, ge | 不等比较 | 无 |
| ==, !=, <=>, eq, ne, cmp | 相等比较 | 无 |
| & | 位与 | 左 |
| \|, ^ | 位或，位异或 | 左 |
| && | 逻辑与 | 左 |
| \|\| | 逻辑或 | 左 |
| .. | 列表范围 | 左 |
| ?: | 条件操作符 | 右 |
| =, +=, -=, *=, | 复合赋值操作符 | 右 |
| , | 逗号操作符 | 左 |
| not | 逻辑非 | 右 |
| and | 逻辑与 | 左 |
| or, xor | 逻辑或 | 左 |

有了上面这些关于优先级和结合性的知识，我们就很容易理解下面的表达式的执行顺序：

```
2 ** 3 ** 4    # 2 ** (3 ** 4), or 2 ** 81, or approx 2.41e24
72 / 12 / 3    # (72 / 12) / 3, or 6/3, or 2
30 / 6 * 3     # (30/6)*3, or 15
```



当你无法确定某操作符是否优先执行时，一定要用括号明确区分出来。用多行、空格等方式提高程序的可读性。

## 5.6 小结与练习

### 5.6.1 小结

本章介绍了各种操作符，为我们熟练运用 Perl 5 扫清了又一个障碍。

第 1 节介绍了数字操作符，包括基本数字操作符、数字比较操作符、赋值操作符和自增自减操作符。这几种操作符我们可能在其他程序语言里也见到过，合理地使用它们可以使程序简洁易懂。不过仍要提请读者注意，使用自增自减操作符时，一定要分清操作符在前面和在后面的情形。

第 2 节介绍了字符串操作符，包括字符串比较操作符、字符串联结操作符和字符串重复操作符。这几种操作符都是比较重要的，尤其是后两种，在其他语言里很少出现，读者一定要掌握它们，因为从一定意义上来说，正是它们体现了 Perl 与众不同的地方。

第 3 节介绍了逻辑操作符和位操作符，这与其他语言基本相同，不再赘述。

第 4 节介绍了逗号操作符和条件操作符，逗号操作符也是 Perl 中特有的，至于条件操作符与其他语言的也基本相同。熟练掌握它们，将使我们的编程风格更趋灵活。

本章最后一节介绍了操作符的优先级和结合性。每个操作符都有自己的结合性，多个操作符放在一起时，就要注意它们的优先级，即谁先运算谁后运算的问题。只有掌握了各种操作符的优先级和结合性，我们才算是真正掌握了操作符，才能不在实际应用中犯错误。

本章集中地介绍了各种操作符，学习了本章，读者应该掌握 Perl 中各种操作符的使用。下一章将介绍 Perl 中的控制结构。

### 5.6.2 习题与思考

（1）给出下列程序的输出结果。

```
$a = 17;
$b = $a + 3;
print $a,"  ",$b,"\n";
$b = $b * 2;
$b = 4 + ($a = 3);
print $a,"  ",$b,"\n";
$b = ($a += 4);
```

```
print $a,"  ",$b,"\n";
$d = ($c = 5);
print $c,"  ",$d,"\n";
$d = $c = 4;
print $c,"  ",$d,"\n";
print ++($foo = '99') ,"\n";
print ++($foo = 'a0') ,"\n";
print ++($foo = 'Az') ,"\n";
print ++($foo = 'zz') ,"\n";
```

（2）写一个两个数相乘的程序，这两个数由运行时输入。
（3）输入一个字符串和一个整数N，将这个字符串重复N次。
（4）写出下列表达式的结果。

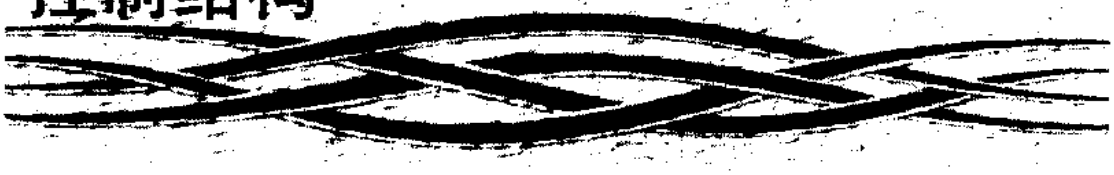
```
print "X" . (4 * 5) ,"\n";
print "fred" x 3, "\n";
print "barney" x (2+1) ,"\n";
$a = 4;
($a % 2) ? ($a += 10) : ($a += 2);
print $a,"\n";
```

（5）写出下列表达式的结果。

```
$a = 123;
$b = 456;
print $a + $b, "\n";
print $a . $b, "\n";
$a = 321;
$b = 4;
print $a * $b."\n";
print $a x $b."\n";
print 2 * 3 * 4, "\n";
print 2 ** 3 ** 4, "\n";
print 2 != 3 != 4, "\n";
print ($b += 2) *= 3, "\n";
print $b += 2, "\n";
```

# 第6章

# 控制结构



控制结构中的有关问题
条件语句
循环语句
其他控制语句
小结与练习



编程人员的基本操作之一是告诉计算机下一步要干什么。控制语句又叫做分支语句，它让计算机做出选择，以便继续执行。控制语句包括 if，if else，if elsif，for，foreach，while，until，last，next，redo 等等。

本章除了介绍 if 系列分支语句和循环语句以外，还将对与它们相关的一些问题，如编程习惯、标识符等进行了讨论。学好了本章，读者就掌握了 Perl 语言的精华。当然其他部分也不可忽略。

本章要点：

- 良好的编程习惯。
- if 系列语句的使用。
- 循环语句的使用。
- last 、next 和 redo 等语句的区别及其灵活运用。
- && 和 || 操作符的灵活使用。

## 6.1 控制结构中的有关问题

正如在其他语言中一样，控制结构在 Perl 程序中也占有举足轻重的地位，几乎没有程序可以避免使用它。本节首先介绍一下与控制结构的使用有关的一些问题。

### 6.1.1 语句块

语句块是由一对大括号“{}”括起来的一连串语句构成的，具有下面的形式：

```
{
    第一行语句;
    第二行语句;
    …… ……
    最后一条语句;
}
```

Perl 会依序执行其中的每一条语句，不过也可以施加控制以改变其执行次序，稍后我们会介绍改变执行顺序的方法。

在文法上，语句块可以放在任何可出现单一语句之处；语句块中可以有 0、1 或多条 Perl 代码；语句块内最后一个语句的分号可以省略。为了方便以后加入新的语句，除非只有一行，否则建议还是加上分号。

### 6.1.2 养成良好的编程习惯

良好的编程习惯不仅有助于提高代码的可读性，而且有助于调试代码。

1. 说明文档

说明语句以表示符 # 开头，它可以从语句中任何位置开始。# 号后面的一切均被解释器忽略。由于说明语句不解释，因此不会减慢代码执行速度。

说明语句对编译代码的价值不是很快就能看出来的，但当你编写了成千上万的代码，你就有可能忘记了以前所写代码的功能及为什么那么写。加上说明语句，就可以省去苦苦回忆某段代码的痛苦。

与不用说明语句相反的错误是滥用说明语句，适量即可，过多则滥。这就是说，别在每一行都加上说明语句。适量使用说明语句和利用含义明确的变量名能使代码一目了然。养成这种习惯后，就可以专心编码，而不必花很多时间去琢磨代码的含义。



说明语句并不是为计算机编写的，而是为自己和其他编程人员编写的，以便于阅读代码。如果说明语句写的很详细，将它们抽出来就可以成为文档了。

2. 缩排

代码缩排能直观地表示每个语句块的开始和结束，这在调试代码时非常有用。一个常见的编码错误是忘记 if 语句的关闭大括号。正确的缩排能帮用户跟踪嵌套代码。

缩排的风格很多，下面是几种常见的格式：

```
if (conditional expression—){
    block of statements}
if (conditional expression—){
    block of statements
  }
if (conditional expression—){
    block of statements
}
if (conditional expression—){
    block of statements
}
```

使用哪种格式并不重要，重要的是始终如一地坚持一种格式。上面第一种是 C 语言中最常用的。有编程经验的程序员都能够深刻体会到缩排的妙处，所以建议读者注意这一点，一开始就不让它错！

读者可以比较一下下面这两种 if 代码块的用法：

```
if ($ready){
    $x++;
```

```
}
if ($ready){
    $x++;
}
```

后者是不是更清晰，更一目了然。

3. 使用标量常量

如果经常要在代码中重复写一些常量，比如π的值 3.1415926，是不是觉得很麻烦，而且写的次数多了难免会出错，再说读程序时不易察觉它们表示的是一个量，也不利于代码中的数值存取相同的精度。使用标量常量就不会出现这个问题了。

大多数语言都有办法让变量名变为常量值，即可以只在声明时对变量名赋值一次。Perl 5 没法让变量名变成常量值，因此需要由编程人员而不是由编译器保证不修改已指定为常量的变量。

使用变量名作为散列的索引是该方法的一个很典型应用。散列中索引很多时，你将无法记住那个位置代表什么值，下面的例子可以很好地解决这一问题：

```
%person = ( UNITID      3456,
            NAME        Li Ping,
            SEX         male,
            AGE         25,
            ADDRESS     Beijing,
            ... ...
)
```

使用这样的写法后，要察看某人的名字只需调用如下语句：

```
$name = $person{NAME};    # $name = Li Ping
```

若用数字作索引，则先要查到名字所在的位置，然后再调用，当系统中存有上千条索引时，这样做几乎是不可能的。

## 6.2 条件语句

### 6.2.1 if 系列语句

与其他语言中相同，在 Perl 中使用 if 语句时，计算机对与该语句相关联的表达式进行求

值，如果条件表达式为真，则计算机执行条件表达式后面的语句块，否则执行真语句块后面的第一条语句。

Perl 究竟是如何判定真假的呢？与其他语言不同，Perl 的判定方法有点奇怪，不过结果仍是我们直觉上所预期的。控制语句会在标量环境下以字符串取值。如果语句是标量，会被转换成字符串：若该字符串是“0”或空字符串(长度为 0)，语句值会被认定为假。除此之外的其他情形都是真。这种判断方法虽然听起来很好玩，不过对系统来说非常方便，因为只要分辨空字符串、“0”和非空字符串即可，不必另外设计一套机制判定数字的真假。下面是一些例子：

```
0       # 转换成“0”，故为假
10 – l0     # 转换成“0”，故为假
3       # 转换成“3”，故为真
“”      # 空字符串，故为假
“l”       # 非空亦非“0”，故为真
“00”      # 不同于“0”，非空亦非“0”，故为真
“0.0000”     # 同上
"0.00" + 0     # the number 0 (coerced by the +), so false.
\$a     # a reference to $a, so true, even if $a is false.
undef()     # 取值为“”，故为假
```

if 系列语句有下列几种：if 语句，if/else，和 if elsif else。这些分支语句符合下列规则：

（1）一个条件表达式只允许一个 if 语句；

（2）if 语句可以有 0 个或 1 个 else 块；

（3）if 语句可以有 0 个、1 个或多个 elsif 从句。

下面分别介绍 if 系列语句。

### 1. if 语句

if 语句可以选择程序的“走向”，如：

```
if (some_expression) {
    true_statement1;
       true_statement2;
         …… ……
   }
   next_statement;
   ……
```

程序执行到这里时，如果 some_expression 为真，则执行 true_statement1、true_statement2、

true_statement3 及大括号内的其他语句；如果 some_expression 为假，程序则跳过括号{}内的语句块，转而直接执行 next_statement，然后继续往下走。

下面是一个简单的例子：

**程序清单 6-1：if 语句的使用**

```
$firstVar  = 20 - 6;
if ($secondVar == 10){
      print "True!\n";
}
print "False!\n";
```

程序输出为：

```
False!
```

2. *if else 语句*

if else 语句的用法如下：

```
if (some_expression) {
 true_statement1;
    true_statement2;
    ... ...
}
else{
    false_statement1;
    false_statement2;
    ... ...
}
next_statement;
.....
```

下面就是一个 if else 语句例子：

**程序清单 6-2：if else 语句的使用**

```
print   " how much is this dictionary?";
 $price = <STDIN>;
 chomp($price);
 if ($price <= 65) {
```

```
        print "So I have enough money to buy the dictionary.";
    }
    else {
        print "So I don't have enough money to buy the dictionary.";
    }
```

程序执行输出结果为：

how much is this dictionary?78

So I don't have enough money to buy the dictionary.

3. if elsif else 语句

如果你想做多重选择，可以使用 elsif 语句。Perl 5 的 if elsif else 语句允许在 if 从句结果为假时，再出现条件表达式。elsif 从句具有多种形式，但和 else 从句一样，elsif 也不能独立存在，只能放在 if 语句之后。其使用方法如下：

```
if (some_expression_one) {
    one_true_statement1;
    one_true_statement2;
}
elsif (some_expression_two) {
    two_true_statement1;
    two _true_statement2;
}
elsif (some_expression_three) {
    three _true_statement1;
    three_true_statement2;
}
... ...
else {
    all_false_statement1;
    all_false_statement2;
  ... ...
}
next_statement;
......
```

每一个表达式都会被依次计算（这里表达式是指 some_expression_one、some_expression_two 和 some_expression_three）。若某个判断语句为真，对应代码块里的每条语句会被依序执行，其他分支判断及其代码块则会被忽略。如果所有判断语句均为假，系统便会执行 else 的代码块(如果有的话)。如果 some_expression_two 表达式为真，则执行 two_true_statement1、two _true_statement2 和 two _true_statement3，然后跳过后面的各个条件判断和 else 块，执行 next_statement 及其以后的语句。

虽然可以不要 else 块，但是通常最好用上它，可以根据需要来增减 elslf 的数目。

**程序清单 6-3：if elsif else 语句的使用**

```
print "Please input the city: \n";
$city = <STDIN>;
chomp($city);
if ($city eq "New York") {
    print "New York is northeast of Washington, D.C.\n";
}
elsif ($city eq "Chicago") {
    print "Chicago is northwest of Washington, D.C.\n";
}
elsif ($city eq "Miami") {
    print "Miami is south of Washington, D.C.  And much warmer!\n";
}
else {
    print "I don't know where $city is, sorry.\n";
}
```

程序输出为：

```
Please input the city:
Washington
I don't know where Washington is, sorry.
```

### 6.2.2 unless 语句

有时，我们“只要”else 块，不过这在语意上有点别扭。所以 Perl 提供了 unless 语句，使用方法如下：

**程序清单 6-4：unless 语句的使用**

```
print  " how much is this dictionary?";
$price = <STDIN>;
```

```
chomp($price);
unless ($price > 65) {
        print "So I have enough money to buy the dictionary.";
}
```

unless 也可以有 else 块，意思和“正面来说”的 if 一样，如下例所示：

**程序清单 6-5：unless 语句的使用**

```
print   "how much is this dictionary?\n";
$price = <STDIN>;
chomp($price);
unless ($price > 65) {
        print "So I have enough money to buy the dictionary.";
}
else {
        print "So I don't have enough money to buy the dictionary.";
  }
```

程序输出为：

```
how much is this dictionary?
90
So I don't have enough money to buy the dictionary.
```

### 6.2.3　if 语句的嵌套

与在其他语言中一样，if 语句可以嵌套在上述各个语句块中。下面是一个例子，当然不是真正的代码，而是一种类似描述语言的伪代码：

```
if(Today is workday) {
    Go to work
}
else {
    if (Today is Saturday) {
       Do the shopping
    }
    elsif (Today is Sunday) {
      Sleep to 10 am.
    }
    watch TV for 1hours
```

```
    go to visit a friend
}
```

对该段描述解释如下：如果今天是工作日，就去上班；如果今天是周末而且是周六，就去购物；如果今天是周末而且是周日，早上就多睡会儿（到 10 点钟）；只要是周末就看一个小时电视，还要去拜访一位朋友。

下面的程序就是读者经常要用到的判断是正数、零还是负数的程序，这里以 if 嵌套形式来写，当然还有其他写法：

**程序清单 6-6：if 语句的嵌套**

```
print "Please input the number : \n";
$number = <STDIN>;
chomp($number);
if ($number > 0) {
    print("plus\n");
}
else {
        if($number < 0){
            print("minus\n");
        }
        else{
          print("zero\n");
        }
}
```

程序执行情况如下：

```
Please input the number :
4
plus
```

## 6.3 循环语句

程序设计员一般都知道，编程工作常常需要一遍遍重复相同的工作，这时人们希望最好有一种机制提供这种功能，免除重复工作的单调与浪费。本节介绍的几种循环语句就可以完

成这一工作。

循环是程序语言的要素之一，它像一个圆，有开始、中点和结束。开始是一个表达式，中间是语句块，结束是闭大括号。开始、中点和结束可以一遍接一遍地重复执行。循环从句可以让计算机多次重复一个操作。

循环首先测试条件表达式，如果条件表达式通过测试，则执行语句块。语句块达到闭大括号时，再次从条件表达式开始。只要条件表达式通过测试，与循环相关联的语句块就继续执行。

### 6.3.1　while/until 循环

在 Perl 中，用户可以用 while 语句制造循环，其用法如下：

```
while (some_expression)
{
    statement_1;
    statement_2;
    ... ...
}
```

程序执行到 while 语句时，首先检查控制语句(some_expression)，若其值为真，就会执行循环体一次，即执行 statement_1，statement_2 和 statement_3。如此不停反复下去，直到控制语句的值为假，Perl 才开始执行 while 循环的下一个语句。请看下面的例子：

**程序清单 6-7：while 循环的使用**

```
$x = 10;
while ($x > 0) {
    print "x is : $x at present\n";
    $x--;
}
```

这个程序执行循环体 10 次，分别对应$x = 10,9,...1 的情形。

下面是一个售票的例子，循环至系统知道票售完为止：

**程序清单 6-8：使用 while 循环的售票系统**

```
$tickets_sold = 600;
while ($tickets_sold < 1000) {
    $available = 1000 - $tickets_sold;
```

```
    print "$available tickets are available. \nHow many would you like: ";
    $purchase = <STDIN>;
    chomp($purchase);
    $tickets_sold += $purchase;
}
print "This show is sold out, please come back later.\n";
```

程序执行情况如下：

```
400 tickets are available.
How many would you like: 300
100 tickets are available.
How many would you like: 200
This show is sold out, please come back later.
```

某些时候，说“直到某事为真”会比“当某事为假”更顺口。Perl 提供的 until 语句正可以做到这件事，其用法如下：

```
until (some_expression) {
        statement_1;
        statement_2;
        ... ...
}
```

上例用 until 语句可以改写为：

**程序清单 6-9：until 循环的使用**

```
   $x = 10;
until ($x <= 0) {
    print "x is : $x at present\n";
    $x--;
}
```

无论 while 或 until，一旦控制语句的值让循环终止，就会完全跳过循环内的语句。以上面的程序来说，若开始给$x 的赋值为 -1，那么循环一次程序便不会执行了。

下面再举一个以数组的长度为循环次数的例子：

**程序清单 6-10：以数组的长度为循环次数的 while 循环**

```
   $count = 1;
```

```
while ($count <= @array) {
    print ("element $count: $array[$count-1]\n");
    $count++;
}
```

Perl 允许用户制造无穷循环，这不但合法，有时也是因为实际应用上的需要，所以不会被认为是错误。例如。写一个程序，如果不发生错误，就让循环不停地重复，并在循环后面写出处理错误的程序，让它不停执行直到系统死机。

### 6.3.2　do while/until 循环

前面所介绍的 while / until 语句在程序进入循环之前首先检查条件语句；如果该条件一开始即为假，那么里面的语句一次都不会被执行到。

以这种情况来说，我们必须在循环最后判断条件，此时就必须用到 do while/until 语句了。该语句的用法和 whlle 很像，只是它至少会执行一次循环体，然后才检查条件：

```
do  {
    statement_1;
    statement_2;
    ... ...
}
while (some_expression);
```

或

```
do {
    statement_1;
    statement_2;
    ... ...
 }
until (some_expression);
```

使用 do 语句有一个很实用的好处：即可以在循环开头将数据初始化。do 语句提供了一种编写整洁可读代码的方式。

若使用 do/while 语句，Perl 会执行 do 块里的语句，最后再检查条件。若条件式为假，循环结束；否则循环会再进行一次。

下面是前面程序清单 6-9 例子的 do/while 形式：

程序清单 6-11：do/while 循环的使用

```
$x = 0;
do {
    print "x is : $x at present\n";
    $x--;
}
while ($x > 0);
```

该程序执行的结果是在屏幕上打印出：

```
x is : 0 at present
```

while 或 until 循环特别适合于处理文件和用户输入输出(I/O)，它能一直执行到发生某个条件为止。

### 6.3.3 for 循环

Perl 另外还有一个循环语句——for，它的用法和 C 语言或 Java 语言中的 for 几乎没什么不同。for 语句主要用于确定次数的循环。其用法如下：

```
for (起始语句；测试语句；步进语句) {
    statement_1;
    statement_2;
    …… ……
}
```

上面的循环也可以用 while 语句改写如下：

```
起始语句；
while(测试语句) {
 statement_1;
statement_2;
…… ……
步进语句；
}
```

系统会首先执行起始语句。一般可在这里为变量赋值，不过这不是硬性规定，读者甚至可以什么都不写，但分号还是要写。测试语句的值若为真，循环体被执行一次，接着执行步进语句。

下面这段程序是用 for 循环改写的程序清单 6-8 的售票程序。

**程序清单 6-12：for 循环的使用**

```
for ($sold = 0; $sold < 10000; $sold += $purchase) {
    $available = 10000 - $sold;
    print "$available tickets are available.  How many would you like: ";
    $purchase = <STDIN>;
    chomp($purchase);
}
```

执行结果与上面的相似。

在 for 循环中的起始及步进语句中，可以使用逗号操作符，以完成多个操作，下面是一个例子：

```
for ($line = <STDIN>, $count = 1; $count <= 3;  $line = <STDIN>, $count++) {
    print ($line);
}
```

它等同于下列语句：

**程序清单 6-13：while/for 循环的等价**

```
$line = <STDIN>;
$count = 1;
while ($count <= 3) {
    print ($line);
    $line = <STDIN>;
    $count++;
}
```

### 6.3.4　foreach 循环

另外一个循环语句是 foreach。它可接受一个列表，将其中的数据作为参数一次一个赋给某标量变量，并执行每一条有效赋值的代码块语句。其用法如下：

```
foreach $i (@some_list) {
    statement_1;
    statement_2;
```

```
    ... ...
}
```

注意，一旦循环结束，标量变量的值会自动回复到循环之前的状态。对 foreach 循环来说，这就好像把它们当成局部变量(local variable)一样。

下面是 foreach 语句的使用例子：

**程序清单 6-14：foreach 循环的使用**

```
print max(45..121, 12..23) . "\n";
print max(23..34, 356..564) . "\n";
sub max {
    my($max) = shift(@_);
    foreach $temp (@_) {
        if $temp > $max
            $max = $temp;
    }
    return($max);
}
```

程序执行的结果为：

```
121
564
```

注意，foreach 所用的列表可以是任何值为列表的语句，不一定要数组变量，甚至还可以不写标量变量，让 Perl 使用默认的变量“$_”。在 Perl 中，许多运算都会以$_做为默认的变量，任何地方都可以用一般的变量取代$_。

举例来说，如果不指定任何值给 print，就会打印出$_的内容，所以上面的程序也可以简写成：

```
foreach (1 .. 10) {
    print;
}
```

是否觉得这样比较容易呢？等到你学会了更多“偷用”$_的语句、操作符或函数，就能把程序写得更简洁一点了。

如果读者在应该使用列表值的地方使用了真正的变量，取代返回列表的函数，那么 Perl 会在循环中把用到的变量视为传入的变量的别名，而不仅仅是拷贝值而已。这样，如果在循环中改变了该标量变量，列表里的对应元素也会被相应改变，这一点读者一定要注意。例如：

```
@x = (1,2,3,4);
foreach $number (@x)
{
    $number += 10;
}
```

程序执行后，数组@a 的值变成了(11,12,13,14)，与原来的不同了。这是 Perl 中的一个特别的设计。

## 6.4　其他控制语句

### 6.4.1　标记修饰符

在 C 语言中如果想一次跳过多个循环，一定要用到难缠的 goto 语句不可。在 Perl 中就比较简单了，只要为想去执行的代码块取个名字，即标记（label），再配合其他控制语句如 redo、next、last 就可以了。

标记有自己的命名空间，和标量变量、列表、hash、函数一样，不过由于它没有特别的前置符号（如标量变量的$、数组的@），所以在命名上会有一些限制：如尽量不要和其他的命名冲突，标记名称最好由大写字母和数字构成，这样保证不会和任何保留字冲突，而且看起来也比较明显。选定标记之后，只要把它放在需要的语句前面，并加上冒号即可，下面是此用法的一个示例：

```
MYLABEL: while (condition) {
                statement;
                statement;
                if (another_condition) {
                    last MYLABLE;
                }
            }
```

如果在 last 之后加了一个 MYLABEL 的参数，这将告诉 Perl 离开名称为 MYLABEL 的代码块，而不是离开最内圈的代码块。

### 6.4.2 修改循环控制流的几个语句

1. last 语句

读者可能会想，Perl 里怎么没有像 C 语言里的 break 语句和 continue 语句呢？有它们将方便多了。事实上，Perl 已提供了功能类似的语句：last。

last 语句会中断最内侧的循环，继续执行紧临区段后的语句。last 语句可以带也可以不带标号修饰符，例如：

```
while (something) {
        something;
          something;
          if (somecondition) {
               otherthing;
               otherthing;
                last;              # 跳出 while循环
          }
          morething;
          morething;
  }
  #  last跳到这里
```

若 somecondition 为真，会执行那些 otherthing，而最后的 last 会强迫循环结束。

使用不带标号修饰符的 last 语句时，退出当前语句块；使用带标号修饰符的 last 语句时，退出与所带标号修饰符相关的语句块。注意，退出的语句块也可能是在当前语句块以上的好几层。

last 语句只会对循环代码块起作用，其他有文法结构的语句代码块并不能用 last 语句中断。总的来说，if / else、以及 do while / until 的代码块不能用 last 来中断，只有 for、foreach、while、until 以及“普通代码块”才可以。所谓普通代码块，是指不属于 if / then / else、循环或函数的代码块。

下面是一个具体的例子：

**程序清单 6-15：last 语句的使用**

```
@array = ("A".."Z");
for ($index = 0; $index < @array; $index++) {
    if ($array[$index] eq "T") {
        last;
    }
```

```
}
print("$index\n");
```

程序输出结果为：

```
19
```



简单来说，last 的意思是退出循环，与 C 语言中的 break 作用相同。

2. next 语句

next 和 last 一样，都会改变程序执行的流程。不同的地方是，next 只是略过循环剩下来的语句，但不会中断循环。其用法如下所示：

```
while (some_condition) {
    something;
    something;
    if (other_condition) {
        otherthing;
        otherthing;
        next;
    }
    morething;
    morething;
}
```

若 other_condition 为真，则执行 something，otherthing 被略过；然后再回到 while 循环，再次判断 some_condition，若它为真，则继续执行循环。需要提醒的是，if 语句代码块在这里是不被视为循环代码块的。

next 语句可以带也可以不带标号修饰符：使用不带标号修饰符的 next 语句时，退出本次执行的当前语句块，继续判断循环条件执行循环；使用带标号修饰符的 next 语句时，退出与所带标号修饰符相关联的循环，仍需要继续判断循环条件执行循环。

下面是一个具体的例子：

**程序清单 6-16：next 语句的使用**

```
@array = (0..9);
print("@array\n");
for ($index = 0; $index < @array; $index++) {
    if ($index == 3 || $index == 5) {
```

```
            next;
        }
        $array[$index] = "*";
    }
    print("@array\n");
```

程序执行结果为：

```
0 1 2 3 4 5 6 7 8 9
* * * 3 * 5 * * * *
```



简单来说，next 的含义是执行下一个循环，与 C 语言中的 continue 作用相同。

3. redo 语句

redo 是本节第三个可以用于跳过循环区段的语句。它可以跳到循环的最前面，但不会再执行一次控制语句。请看下面的例子：

```
SOMELABEL:
{   # redo跳到这里
    something;
      something;
      if (other_condition)
      {
          otherthing;
          otherthing;
          redo SOMELABEL;
      }
      morething;
      morething;
}
```

如果 other_condition 为真，则执行 otherthing，然后跳到 something；如此时 other_condition 为假，则执行 morething，然后跳出 SOMELABEL 语句块。

redo 语句也可以带或不带标号修饰符：不带标号时，redo 语句跳到所在语句块的第一条语句上；带标号时，跳到与所带标号相关联的语句块的第一条语句上。

下面是使用 redo 语句的一个具体例子：

**程序清单 6-17：redo 语句的使用**

```
print("What is your name? ");
$name = <STDIN>;
chomp($name);
if (! length($name)) {
    print("Msg: Zero length input. Please try again\n");
    redo;
}
print("Thank you, " . uc($name) . "\n");
```

程序执行情况如下：

```
What is your name? Tom Jackson
Thank you, TOM JACKSON
```



redo 是 Perl 的一个特有命令，其含义是重复此次循环，即循环变量不变，回到循环起始点。但要注意，redo 命令在 do 循环中不起作用。

4. goto 语句

读者可能在其他语言中见过 goto 语句，它可以使程序直接跳转到标号处。由于使用 goto 语句会使程序在逻辑上显得很混乱，所以 Perl 虽然提供了对 goto 语句的支持，但是并不推荐使用它。实事上，因为 Perl 已提供了大量的循环以及控制语句，如 last、next、redo 等，读者大可不必使用 goto，就可以完成相同的功能。

### 6.4.3　表达式修饰符

Perl 提供了另外一个“若 A 则 B”的表示方法。可以在一般的 if 语句中加上修饰语，即把条件表达式放在后面，像这样：

```
some_statements if condition;
```

以这个语句为例，若 condition 为真，则执行 some_statements 部分。这样的写法与下面的写法相同：

```
if (condition){
    some_statements;
}
```

下面是一些类似的用法：

```
expression unless condition;   # 同unless (condition)   {expression;}
expression while condition;    # 同while (condition)    {expression;}
expression until condition;    # 同until (condition)    {expression;}
```

上面这些语句都将先执行 condition，视其真假决定是否执行 expression。



修饰语必须是简单表达式，不能是语句代码块。

这种写法虽然必须倒着读，不过在许多时候却相当好用。例如，下面的程序片段：

```
print ("This is zero.\n") if ($var == 0);
print ("This is zero.\n") unless ($var != 0);
print ("Not zero yet.\n") while ($var- - > 0);
print ("Not zero yet.\n") until ($var- - == 0);
```

这样写是不是比较直观呢？而且用英文读起来也比较自然。



虽然条件判断写在后面，但却是先执行的。

### 6.4.4 &&和||

到目前为止，“若 A 则 B”可以用下面的两种方法表示：

```
if (this)  { that;}
that if this;
```

其实，Perl 中还有第三种方法：

```
this && that;
```

上面这个语句看起来一定会觉得很别扭，&&不是逻辑操作符吗？为什么会是这个意思。要回答这个问题，先看看下面的讨论：

（1）this 为真

由于 that 尚未执行，整个语句的值仍未知，为了要执行完整的语句，that 就会被执行；

（2）若 this 为假

整个语句的值就一定为假，就不需要再执行 that 语句了，所以 that 不会被执行。

因此，this && that 成立。用同样的想法，读者应该可以得出 || 与 unless 同样功能的结论，比如：

```
unless (this)    { that; }
```

与下面语句的效果相同:

```
this || that;
```

下面是使用它们的一个具体例子：

**程序清单 6-18：&&和 || 使用**

```
@a = (97..122);
@b = (a..z);
@a == @b && print "They are equal.";
@a == @b || print ("They are not equal.");
```

程序的逻辑关系如下：

- 表@a 等于列表@b，则第三条语句前部为真，继续执行&&后面的语句，输出"They are equal."；第四条语句就不执行了。
- @a 不等于列表@b，则第三条语句前部为假，因此不再执行&&后面的语句；第四条语句，因为前面为假，所以还需执行后面的语句，输出("They are not equal.")。

程序执行结果为：

```
They are equal.  # 说明26个字母与其相应的ASCII码是相等的。
```

到底何种写法较好呢?其实每个人的看法是不一样的。我们唯一能给的建议是，把语句中最重要的部分尽量放在前面。照这个原则写程序，无论如何都是正确的。

## 6.5　小结与练习

### 6.5.1　小结

本章介绍了 Perl 中的各种控制结构，控制结构是程序整体上的框架，好的控制结构，可以使程序看起来条理清楚，也大大减少了出错的可能性，因此学好这一章十分重要。

第 1 节介绍了与控制结构有关的几个问题，包括语句块和良好的编程习惯。语句块是指完成一定功能的代码序列，它是控制结构中的重要元素。良好的编程风格是编写和调试程序所不可缺少的，也可以使逻辑分明的控制思路得以清晰的体现。

第 2 节介绍了条件语句，它几乎是每一种编程语言中都有的一种结构。条件语句包括：

if 系列语句（即 if 语句、if /else 语句和 if /elsif/ else 语句）、if 嵌套语句和 unless 语句。前几种在其他语言中读者可能都已经见过了，它们的用法也与其他语言中基本相同。只有 unless 语句是 Perl 中比较特殊的，它是为了语义上的方便而引入的，但完全可以用 if 语句来替代。

第 3 节介绍了循环语句，它是为了重复执行相同的动作而设计的。其中 while 是首先判断条件，在条件满足时再执行一定的语句块；do/while 语句是先执行语句块，然后再判断条件；上面两种循环都可以用 until 语句来改写。until 语句和 unless 语句一样，也是为了自然语言上的方便，它的功能与 while 语句完全相似，只不过在逻辑关系上有所不同。如果知道循环的次数，还可以使用 for 循环语句。foreach 循环是专门用来对数组和列表进行处理的，有了它，对数组元素的存取和访问将变得十分方便。

第 4 节介绍了其他的控制语句，标号修饰符可以表示一段特定的语句块。last、next、redo 和 goto 语句可以方便地改变循环控制流的走向，熟练掌握它们，将使我们对程序具有完全的控制权。但是最好不要用到难缠的 goto 语句，因为其他语句完全可以替代它的功能。

本节的最后阐明了表达式修饰符可以使程序读起来更像自然语言，而用&&和||的形式组织起来的语句将使程序具有最高的执行效率。

学习了本章，读者应该掌握 Perl 中各种控制结构的使用。掌握了控制结构，就可以写出功能完备而且逻辑清晰的 Perl 程序了。下一章将介绍 Perl 中的正则表达式。

### 6.5.2 习题与思考

（1）写一段程序，读取当天的气温，判断天气是太热（气温高于 30 度）、太冷（气温低于 10 度）还是适中（气温介于二者之间）。

（2）写一段程序，读入一连串的数字（每行一个），直至读入数字 999 结束，计算它们的和，注意 999 不要计算在内。

（3）写出下面程序的输出结果。

```
@array = (0..5, 5..10);
print("@array\n");
foreach (@array) {
    $_ = "**" if ($_ == 5);
}
print("@array\n");
```

（4）写一个程序，输入一连串的字符串，每行一个，然后将它们颠倒次序输出，程序中不允许使用 reverse 语句。

（5）写出下面程序的输出结果。

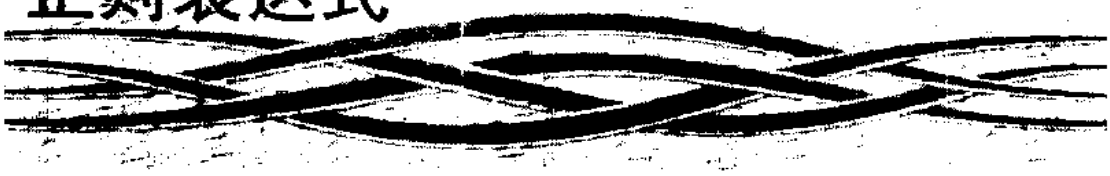
```
@ary1 = (0 .. 9);
@ary2 = (5 .. 10);
```

```
print @ary1,"\n", @ary2,"\n";
for ($i = 0; $i < @ary1; $i++) {
    for ($j = 0; $j < @ary2; $j++) {
        if ($ary1[$i] > $ary2[$j]) {
            next;
        }
        $ary1[$i] += $ary2[$j];
    }
 }
foreach $num(@ary1) {
    print $num,"  ";
}
print "\n", @ary2,"\n";
```

（6）分别用使用列表和不使用列表两种方式编写一个程序，输出 1 到 20 之间的自然数的平方。

# 第7章

# 正则表达式



正则表达式简介
正则表达式的构成
正则表达式的正则
模式修饰符
与正则表达式相关的其他问题
小结与练习



正则表达式和模式匹配是利用 Perl 5 功能的基础。Perl 5 是进行字符串操作和模式匹配的最理想最简便的语言，也是广受欢迎的 Internet 编程语言之一。

本章要点：

- 正则表达式中的元字符。
- 正则表达式的使用正则。
- 正则表达式的应用。
- 模式修饰符。



# 7.1 正则表达式简介

## 7.1.1 什么是正则表达式

“正则表达式”（regular expression，也称作正则表达式）是一种字符串模式，可用来和字符串进行匹配。模式指在字符串中寻找的特定序列的字符，由反斜线包含：/def/即模式 def。我们常常会依一般语句符合的位置和情形，取得字符串中的某些部分，或把符合要求药酒的某些部分以其他字符串取代。

许多程序，包括 VINX 命令、sed、awk、ed、vi、emacs，甚至包括许多种 shell，都提供有多种相同或不同的通配符，用以进行正则表达式的字符串匹配功能。Perl 的功能是上面这些程序的“超集”——也就是说，任何上述程序提供的功能，Perl 都能提供，只是操作符的写法有所不同。

正则表达式可以处理用户输入、匹配字符串、操作文件 I/O 等等。正则表达式是元字符、正则字符和匹配计数器的字符串，匹配计数器可生成模式。模式将字符串的一部分或全部进行匹配。遇到在代码中使用的正则表达式一般是由一个原子（atom）操作构成的，以此为基础构成复杂的模式。

## 7.1.2 一个简单的例子

刚接触正则表达式概念的读者可能有点迷惑，到底什么是正则表达式，它到底有什么用途？下面就是一些简单的用法，以帮助读者建立感性认识。

**程序清单 7-1：使用正则表达式的一个简单例子**

```
if ($question =~ /please/) {
    print ("Thank you for being polite!\n");
}else {
    print ("That was not very polite!\n");
}
```

在 Perl 中，读者可在“please”　的前后两端用斜线标记，以代表斜线之间的内容是个正则表达式。上例中的“=~”、“!~”是匹配操作符。“=~”检验匹配是否成功：若在所给字符串中找到了要求的模式，则返回非零值，即 true；不匹配则返回 0，即 false。!~则相反。

## 7.2　正则表达式的构成

模式可以由构成正则表达式的元字符或简单字符构成。

用喜欢的文本编辑器搜索某个单词时，通常输入简单的字符（比如 she），文本编辑器的搜索功能从文本文档中寻找每个出现 she 的地方。

如果用模式匹配搜索 she ，则语句如下：

```
if (/she/i) {
print "Find a match!\n";
}
```

这样 Perl 5 就会按顺序寻找字符 s、h 和 e 。这个模式匹配 SHE、she 和 shell 等，这是因为语句修饰符末尾的模式修饰符“i”使 Perl 5 在进行匹配时忽略大小写。i 和其他模式修饰符将在本章后面介绍。

利用文本编辑器可以指定整字匹配，避免在寻找 she 时匹配 shell 等，此外还有许多方法可以让 Perl 5 只查找 she 而不找 shell。让 Perl 5 只进行整字匹配或匹配开头和结尾的方法要使用正则表达式中的元字符。

### 7.2.1　元字符

元字符可以方建立正则表达式，用以匹配所要的字符，而不必一一列出所有不匹配的字符。表 7-1 列出了元字符及其含义，具体用法将在后面介绍。

表 7-1　　元　字　符

| 元 字 符 | 含　义 |
|---|---|
| \ | 转义，不解释后面的元字符 |
| \| | 或，即匹配两者之一 |
| () | 生成一个表达式或原子 |
| {} | 定义原子的最大/最小重复次数 |
| * | 匹配原子 0 次或多次 |

续表

| 元 字 符 | 含　义 |
| --- | --- |
| + | 匹配原子 1 次或多次 |
| ? | 匹配原子 0 次或 1 次 |
| ^ | 匹配原子，在字符串开头 |
| $ | 匹配原子，在字符串末尾 |
| [] | 匹配其中的原子之一 |
| . | 匹配任一字符 |
| \A | 元字符^的替换 |
| \Z | 元字符$的替换 |

### 7.2.2 原子

原子是正则表达式的基本组成部分。简言之，原子就是模式中的一个单个项目。元字符不是原子，它要与原子配对后才有意义。

原子可以是下列项目：

（1）个字符或数字：A~Z，a~z，0~9。

（2）义元字符。

（3）转义特殊字符。

（4）八进制码、十六进制码和控制码，如\o77、\xFF 或\cD。

（5）括号中的正则表达式。

（6）前一模式匹配的反引用。

## 7.3 正则表达式的正则

建立正则表达式时，就是组合原子和元字符串，这些原子和元字符串指定要匹配的模式。下面将介绍建立正则表达式的正则，并举例说明。

一个正则表达式就是一种模式（pattern）。模式的某些部分会用某种形式和字符串中单独的一个字符匹配，而其他部分会和字符串中一个以上的部分匹配，也被称为单字符类型和多字符类型。下面我们将分别讨论单字符类型和多字符组合类型。

### 7.3.1　单字符类型

单字符类型是最常见及最常用的类型。它的意思是，如果我们在正则表达式中放入一个1，字符串对应的地方也要出现一个 1 才算符合匹配条件。

次常见的单字符类型是英文句号“.”。除了换行符“\n”，任何字符都可以和这个实心句号匹配。例如，/b./ 会和任何长度为二，且开头为 b 的字符串匹配，但“b\n”除外。

另外还有一种类型称为字符类(character class)，写法是左右两个中括号“[ ]”，中括号中间放入字符。它的意思是，字符串对应此类型的地方，若出现唯一一个符合中括号里列出的字符，就算符合。举例来说：

```
/[12345]/
```

会和任何仅由上面五个数字字符之一所构成的字符串匹配。而下面这个类型：

```
/[aeiouAEIOU]/
```

则会和任何仅含五个元音字母之一的字符串匹配。若想在中括号内再放“[”或“]”字符，必须在前面加上转义字符反斜线“\”；如果只想置入右中括号，把它放在第一个字符的位置也可以。如果想表示一段范围，有个简写的方法，即左右各写始末字符，中间放个破折号（-）即可。如果想表示破折号字符，同样必须在前面加上反斜线“\”。下面是一些例子：

```
[0-9]       # 同 [0123456789]
[0-9\-]     # 匹配0到9之一或破折号“-”
[a-z0-9]    # 匹配小写字母或0到9之一
[a-zA-Z0-9]   # 匹配大小写字母或0到9之一
[a-zA-Z0-9_]   # 匹配大小写字母或0到9之一或下划线“_”
[0-9\[]     # 匹配0到9之一或左中括号“[”
[]0-9]      # 匹配0到9之一或右中括号“]”
```

也可以用“反例”表示字符类别，只要在原来的写法前面加上“^”即可。它的意思是任何不在此类别内的单一字符才算匹配：

```
[^0-9]      # 数字以外的所有字符才符合
[^a-zA-Z0-9]   # 数字和字母以外的所有字符才符合
[^\^]       # “^”以外的所有字符才符合
```

为了使用方便，Perl 预先定义了一些字符类别，现在详细列出如表 7-2：

表 7-2　　预先定义的字符类别

| 字　符 | 类　别 | 反　例 | 反例类别 |
|---|---|---|---|
| \d | [0-9] | \D | [^0-9] |
| \w | [a-zA-Z] | \W | [^a-zA-Z] |
| \s | [\r\t\n\f] | \S | [^\r\t\n\f] |

## 7.3.2　组合类型

单字符类型毕竟受到了局限性，最能看出正则表达式威力的地方是把它们组合起来用。在这一节里，我们将介绍类型组合（grouping pattern）的用法。

1. 序列

首先介绍的类型组合是序列。什么是序列呢？举例来说，正则表达式“abc”就是一个序列，它的意思是，如果字符串某部分有个 a，后面跟一个 b，再跟一个 c，这样操作符合。

下面是一个例子，它匹配输入的字符串，如果含有 ok，就输出：

**程序清单 7-2：正则表达式中的序列**

```
while (<>) {
  if (/ok/) {
    print $_;
  }
}
```

程序执行情况如下：

```
abc
ok
ok # 输出
thankok
thankok    # 输出
all
book
book    # 输出
okk
okk    # 输出
```

2. 重复符号

在前面介绍过的星号 * 也是一种类型组合。它代表*号前面一个字符，若出现零次以上，就操作符合。加号 + 的意义和星号很像，不过它代表符号前面一个字符必须在目标字符串内出现一次以上才操作符合。另外，问号 ？ 的意义也很类似，代表符号前面的一个字符必须恰好出现零次或一次。

如果只有上面三种组合方法，要表示“5 到 10 个 x”，必须写成“x?x?x?x?x?”。但这看起来很累赘。解决的办法是利用“一般重复符号”，它的写法是在一对大括号中放一个或两个数字，还要有一个字符放在左括号前面。

这样，“5 到 10 个 x”就可以写成

```
/x{5,10}/
```

如果不写第二个数字，如/x{5,}/，就代表‘重复 5 次以上”；如果连逗号也不写，/x{5}/，意思是“刚好 5 个 x”。

这样，/a.{5}b}/的意思就是“以 a 开头，b 结尾、中间隔上 5 个非换行字符”的字符串，中间 5 个字符不一定要相同。

事实上，一般重复符号完全可以取代 * 、+和？这些符号。不过由于打字上的方便，这些符号仍然继续被使用。

正则表达式若出现两个重复符号，必须遵守“最左边最贪心”的正则。请看下面的例子：

```
$_ = "a xxx c xxxxxxxx c xxx d";
/a.*c.*d/
```

以这个表达式为例，第一个“.*”就会和第二个 c 之前所有的字符匹配。读者可能会想，如果让它“只符合到第一个 c”，整个正则表达式就可以完全匹配到了。到目前为止，这并不构成问题，反正这个语句和字符串无论如何都匹配，不过读者稍后就能体会到这件事的重要性。

我们可以在重复符号后面加上问号。让它变得“不贪心”：

```
$_ = "a xxx c xxxxxxxx c xxx d";
/a.*c?.*d/
```

这里的“a.*c?”会和最少的 a、c 之间字符匹配，而不是所有字符——所以会匹配到第一个 c 为止。

下例中我们把字符串和正则表达式再改一下：

```
$_ = "a xxx ce xxxxxxxx ci xxx d";
```

/a.*ce.*d/

在这个例子中，如果“.*”尽可能和第二个 c 之前所有字符匹配，正则表达式里的“e”就不会和字符串的下个字符“i”匹配了。其实如果遇到这种情形，重复符号会自动“重试”，在较前面的地方“中止”匹配。以这个例子来说，会匹配到第一个 c 为止。在更复杂的正则表达式中，重试步骤可能会进行多次，让执行速度下降。以这个例子来说，其实只要加个“?”号就能让 Perl 少做很多事，请读者特别注意这一点。

3. 选项符号

另外还有一种组合符号称为选项符号，如“a| b| c”。这个写法的意思是，只要有一个选项符合（本例里是 a 或 b 或 c）就操作符合。其实选项如果只有一个字符，这种写法还不如 / [abc] / 来得方便。它存在的理由在于，选项可以不只一个字符，如/orange|banana/。

4. 括号

前后包围任何类型的括号对是另外一个组合符号。括号不会改变类型匹配的情形，但可以把匹配的部分记录下来，留待以后参考。

要调用“记住”的部分字符串，写法是在反斜线后面加个数字。该类型的意思是，它代表稍早在前面第某对括号内匹配的部分。例如：

/ orange (.) banana\1/ ;

这个表达式表示系统首先和开头有“orange”字符串，后面跟着任意一个非换行字符，然后是“banana”字符串，最后再跟一个和“orange”、“banana”中间那个字符一样的字符串匹配。比如，这个正则表达式会和“orange&banana&”匹配，而不会和“orange&bananal”匹配。对下面这个正则表达式来说，这两个字符串就都会匹配了：

/ orange . banana./ ;

现在应该可以明白正则表达式中“\1”的意思了吧。它代表正则表达式中，第一个用括号括起来的部分。如果括号对不只一个，第二个部分可以用\2 表示，再后来的可以用\3，\4……，依此类推。举例来说：

比如：

/ \d{2}([\W])\d{2}\1\d{2}/

可以匹配：

12-05-92

注意：/\d{2}([\W])\d{2}\1\d{2}/ 不同于/(\d{2})([\W])\1\2\1/ ，后者只匹配形如 17-17-17 的字符串，而不匹配 17-05-91 等。

还可以在括号中放入一个以上的字符，例如：

```
/a(.*)b\1c/;
```

下面是组合类型匹配的一个综合例子：

```
[0123456789]    # 匹配任何数字
[0-9]    # 同上
[0-9\-]   # 匹配数字 0-9 或负号
[a-z0-9]   # 匹配任何数字和小写字母
[a-zA-Z0-9_ ]    # 匹配任何数字和大小写字母
abc*   # 匹配诸如 ab, abc, abcc, abccc, abcccc...的字符串
(abc)*   # 匹配诸如 "", abc, abcabc, abcabcabc, and so on
^x|y    # 匹配开头为x 中间包含 y的任一行字符
^(x|y)   # 匹配开头为x 或 y的任一行字符
a|bc|d   # 匹配a, bc或 d
(a|b)(c|d)    # 匹配ac, ad, bc或 bd
(song|blue)bird    # 匹配songbird或 bluebird
```

### 7.3.3　定位类型

定位类型有好几种。Perl 拿类型和字符串做对比时，读者可以想像它把类型的头叠在字符串的最前面开始往后扫描，寻找可能的第一次符合。利用定位类型，可以确保类型对正字符串的某些部分。

首先介绍定位类型\b。此类型表示目标字符串在这个地方必须是文字边界，在这之前必须符合其他的条件。请看下面的例子：

```
/fred\b/;    # 会和fred匹配，不会和Frederick匹配
/ \bmo/      # 会和moe和mole匹配，不会和Elmo匹配
/ \bFred\b/   # 会和Fred  匹配，但Frederick或alFred都不匹配
/ \b\+\b/    # 会和 "x+y" 匹配，不会和 "++" 或 "+" 匹配
/abc\bdef/;    # 任何字符串都不可能匹配（因为该处不可能有边界）
```

另外一个定位类型\B 所在之处则要求不是文字边界。例如：

```
/\bFred\B/;     # 会和Frederich匹配，但Fred Flintstone不匹配
```

接下来要介绍的两个定位类型，必须附带一个单字符类型，代表字符串开头或结尾的前一个字符。“^”会和字符串的“开头”匹配——如果该处做为字符串开头确实有意义的话。

另外一个定位类型“$”和“^”很类似，不过它代表的是字符串结尾。若把它放在正则表达式其他的部分，会被视为标量值解析符号。如果真想表示“$”这个字符，必须在它前面加反斜线。

Perl 还支持其他许多定位类型，如\A、\Z 等等，这里不再赘述。后面将会举例说明它们的应用。

### 7.3.4 优先级

正则表达式里面的类型也会有优先级的问题。想想看，“a|b*”到底是什么意思呢?到底是“任意个 a 或任意个 b”，还是“1 个 a，或任意个 b”？

操作符优先级有高低，组合类型和定位类型的优先级也有高低。优先性由高至低的各式类型列在表 7-3 中。

表 7-3　　各种类型的优先级（由高到低）

| 符　号 | 名　称 |
|---|---|
| () (?:) | 括号 |
| + * ? {m,n} ?? +? {m,n}? | 重复操作符 |
| abc ^ $ \A \Z (?=) (?!) | 序列和定位符 |
| \| | 选项符 |

### 7.3.5 正则表达式正则的应用举例

前面我们已经介绍了正则表达式的组成和各种正则，下面给出一些应用的例子，以帮助读者更好的理解和掌握正则表达式。

1. 字符 *和?

前面已经介绍过，* 号匹配 0 个、1 个或多个相同字符；？号匹配 0 个或 1 个该字符。如/de*f/匹配 df、def、deeeef 等；/de?f/匹配 df 或 def。

现在将上面见过的例子改一下，就可以变成匹配 a 和 c（包括 a 和 c）之间含有零个或多个 b 的字符串：

**程序清单 7-3：字符“*”的使用**

```
while (<>) {
  if (/ab*c/) {
    print $_;
  }
}
```

2. 字符 +

它与 * 类似，区别在于 + 意味着一个或多个相同的字符，如/de+f/匹配 def、deef、deeeeef 等。它匹配尽可能多的相同字符，如/ab+/在字符串 abbc 中匹配的将是 abb，而不是 ab。

现在将上面的例子再改一下，就可以变成匹配 a 和 c（包括 a 和 c）之间至少含有 1 个 b 的字符串：

**程序清单 7-4：字符“+”的使用**

```
while (<>) {
  if (/ab+c/) {
    print $_;
  }
}
```

3. 字符 [ ]和[^]

[ ]意味着匹配一组字符中的一个，如/a[0123456789]c/将匹配 a 加数字加 c 的字符串。与+联合使用，例如：/d[eE]+f/匹配 def、dEf、deef、dEdf、dEEEeeeEef 等。^表示除其之外的所有字符，如：/d[^deE]f/匹配 d 加非 deE 字符加 f 的字符串。

为加深理解，下面再给出几个该类型匹配的例子：

```
[^0-9]  # 匹配任何单个的非数字字符
[^aeiouAEIOU]  # 匹配任何单个的非元音字母的字符
[^\^]  # 匹配任何单个字符，除了符号^
```

下面是一个具体的例子：

程序清单 7-5：字符“ [ ]“和“ [^] ”的使用

```
while($answer = <>)
{
   chomp($answer);
   if ("SEND"  =~ /^\Q$answer/i) {
      print "Action is send\n"
   }
   elsif ("STOP"  =~ /^\Q$answer/i) {
      print "Action is stop\n"
   }
   elsif ("ABORT" =~ /^\Q$answer/i) {
      print "Action is abort\n"
   }
   elsif ("LIST"  =~ /^\Q$answer/i) {
      print "Action is list\n"
   }
   elsif ("EDIT"  =~ /^\Q$answer/i) {
      print "Action is edit\n"
   }
   else{
      print "Unknown action\n"
   }
}
```

程序接收输入的命令，然后输出相应的信息：

```
send
Action is send
stop
Action is stop
abort
Action is abort
list
Action is list
edit
Action is edit
end
Unknown action
```

4. 使用转义字符

如果用户想在模式中包含通常被看作特殊意义的字符，须在其前面加斜线“\”。如 *即表示字符 * ，而不是上面提到的一个或多个字符的含义。

表 7-4 是一个转义字符表，其中没有包括表 7.2 中已经涉及到的内容。

表 7-4　　转义字符表

| 代　码 | 含　义 |
|---|---|
| \a | 警报 |
| \b | 字边界 |
| \B | 非字边界 |
| \e | 转义 |
| \f | 进纸 |
| \n | 新行 |
| \r | 回车 |
| \t | 制表符 |
| \oNN | 八进值 |
| \xNN | 十六进制 |
| \cC | 控制符 |

5. 定位符的应用

下面是几个使用定位符的例子：

例 1：/^def/只匹配以 def 打头的字符串，/$def/只匹配以 def 结尾的字符串，结合起来的/^def$/只匹配字符串 def。

例2：程序清单7-6中的程序就是利用定位符检验变量名的类型，看它是否是合法的标量名、数组名、散列名或程序清单7-6中文件名，否则就是非法名称：

**程序清单 7-6：定位符的使用**

```
print "Enter a variant name : ";
$varname = <STDIN>;
while ($varname) {
    chomp($varname);
    if ($varname =~ /^\$[A-Za-z][_0-9a-zA-Z]*$/) {
        print ("$varname is a legal scalar variable\n");
    }
```

```
    elsif ($varname =~ /^@[A-Za-z][_0-9a-zA-Z]*$/) {
        print ("$varname is a legal array variable\n");
    }
    elsif ($varname =~ /^%[A-Za-z][_0-9a-zA-Z]*$/) {
        print ("$varname is a legal hash variable\n");
    }
    elsif ($varname =~ /^[A-Za-z][_0-9a-zA-Z]*$/) {
        print ("$varname is a legal file variable\n");
    }
    else {
        print ("I don't understand what $varname is.\n");
    }
    print "Enter another number : ";
    $varname = <STDIN>;
}
```

程序执行情况如下：

```
Enter a variant name : bbs
bbs is a legal file variable
Enter another number : %hash
%hash is a legal hash variable
Enter another number : $var
$var is a legal scalar variable
Enter another number : _jkdi3 8d
I don't understand what _jkdi3 8d is.
Enter another number : 9aikd
I don't understand what 9aikd is.
Enter another number : a(idk)
I don't understand what a(idk) is.
Enter another number : q_9dike
q_9dike is a legal file varlable
Enter another number : i a
I don't understand what i a is.
Enter another number : @array
@array is a legal array varlable
Enter another number :
```

## 7.4 模式修饰符

正则表达式可繁可简，上节介绍的就是几种简化的正则表达式。模式修饰符不仅能够简化正则表达式，而且可以使正则表达式更有效，甚至返回字符串中所有出现的模式。需要注意的是，模式修饰符并不是正则表达式的一部分，而是添加在正则表达式中尾部的分隔符"/"之后的部分。模式匹配符将影响整个正则表达式。例如，可以改变正则表达式，使其忽略大小写、匹配整个输入字符串而不是只找第一个匹配。

模式修饰符共有六个：

g —— 全局匹配
i —— 忽略大小写的匹配
m —— 多行匹配
s —— 单行匹配
o —— 单次匹配
x —— 多余空格

下面就分别对它们进行介绍。

### 7.4.1 全局匹配

全局匹配模式修饰符"g"用于修改正则表达式输入字符串的次数。正则表达式缺省时，寻找最长的单模式匹配，然后停止。全局匹配可以让正则表达式匹配整个输入字符串。和许多 Perl 5 操作符一样，全局匹配的作用随上下文变化而改变，如：

- 表上下文中的全局匹配修饰符返回每个模式实例；
- 标量上下文中的全局匹配修饰符对每个模式实例返回 true。

使用列表上下文中的全局匹配修饰符时，正则表达式返回每个模式实例。列表上下文中的全局匹配修饰符用法如下：

```
@list = ( $string =~ /pattern/g)
```

每个模式实例保存在数组@list 的一个单元中。利用这种格式可以捕获每个模式实例和计算总实例个数。数组@list 包含每个模式。在标量上下文中，数组包含模式实例的个数，如下例：

```
$number = @list;
```

也可以将要保存的字符串部分放在括号中，只捕获匹配的模式中的一部分。括号中发生的每个模式都会返回，而括号外的内容作为正则表达式的一部分，只有括号中匹配的每个模式实例被返回并保存。

标量上下文中使用全局匹配修饰符时，它一次一个匹配地遍历输入字符串。每个匹配的模式返回 true。下次再匹配输入字符串时，全局匹配修饰符将处理上次匹配后面的一部分。如果只求匹配的次数，可以使用下面的语句：

**程序清单 7-7：全局匹配**

```
$count = 0;
while ($string =~ /pattern/g) {
    $count++;
}
```

变量$string 从左向右搜索模式，每次找到模式时，返回真值。如果$string 中有十个模式实例，则代码十次经历 while 循环。

使用“g”选项后会匹配所有可能的模式，比如：

```
@matches = "sentence" =~ /.e/g;     # 现在 @matches = ("se", "te", "ce")
```

上例也可用匹配的循环：

```
while ("sentence" =~ /.e/g) {
    $match = $&;
    print ("$match\n");
}
```

结果为：

```
se
te
ce
```

下面是另外一个例子：

**程序清单 7-8：使用全局匹配**

```
$digits = "123456789";
@nonlap = $digits =~ /(\d\d\d)/g;
@yeslap = $digits =~ /(?=(\d\d\d))/g;
```

```
print "Non-overlapping:   @nonlap\n";
print "Overlapping:       @yeslap\n";
```

程序输出结果为：

```
Non-overlapping:  123 456 789
Overlapping:      123 234 345 456 567 678 789
```

### 7.4.2 忽略大小写的匹配

忽略大小写修饰符“i”将改变模式中的每个字母原子，使其匹配大写和小写字母。忽略大小写修饰符特别适合于文本搜索。例如，如果要寻找单词 cat 的所有模式实例，可以用下面的语句：

```
/[Cc]at/
```

该模式寻找以大写或小写的 C 字母开头的每个 cat 模式实例。也可以用下面的语句：

```
/cat/i
```

这样不仅简化了代码，而且能匹配 cat 的任意大小写拼写，包括 cAt、caT 或 CAT 等。

忽略大小写修饰符在获取用户交互输入时非常方便，特别是要求输入 Yes/No 响应的问答。对要求输入 Yes/No 响应的问答，可以用下列模式：

```
/(Y(es)? | (N(o)?)/i
```

利用这个模式，用户可以输入任意大小写的 Y 或 N，如果用户输入完整的 Yes、yes 或 YES，这个模式也可以接受。

下面就是一个例子：

**程序清单 7-9：忽略大小写的匹配**

```
print "Input a number : ";
$number = <STDIN>;
while ($number) {
        chomp($number);
        push @array,$number;
    print "Do you want to input another number?   ";
    if (<STDIN> =~ /Y(es)?/i) {
```

```
            $number = <STDIN>;
        }
        else{
                last;
        }
    }
    foreach $number(@array){
            print $number,"   ";
    }
```

该程序要求用户输入数据，输入完后系统询问，如果回答的是 Y 或 y，就继续接受输入，并将输入的数字存入一个数组中。最后输出该数组，程序执行情况如下：

```
Input a number : 78
Do you want to input another number?   y
89
Do you want to input another number?   y
0
Do you want to input another number?   y
-90
Do you want to input another number?   y
0.98
Do you want to input another number?   n
78   89   0   -90   0.98
```

### 7.4.3 多行匹配与单行匹配

处理 HTML 文本区标志中的用户输入时，数据可能是单行或多行。所有数据都存放在一个与文本区标志相关联的变量中。利用模式修饰符“m”和“s”,可以预处理用户输入，找出新行符，也可以让正则表达式忽略$searchString 中的新行符。

要忽略搜索字符串中的新行符，用下列语句：

```
/pattern/s
```

即表示输入的为一个整行（单行）。

下面是一个例子：

```
/a.*bc/s
```

表示匹配字符串 axxxxx \nxxxxbc，但表达式/a.*bc/则不匹配该字符串。
如果输入为多行，要将输入变为相同行格式的 HTML 输入，用下列语句：

```
$input =~ s/ \n/<BR>/gs;
```

这样，每个新行符都变成 HTML 新行中断命令。如果要把输入字符串变为多行字符串，用下列语句：

```
/pattern/m
```

这样就可以寻找到已嵌入新行符的行头和行末，例如：

**程序清单 7-10：多行匹配**

```
$input = "This \nis \none \nscalar \n\object.\n That has multiple lines\n";
print "Matched object\n" if $input =~ m/^object/m;
print "Matched scalar\n" if $input =~ m/scalar$/m;
print "Matched object\n" if $input =~ m/^scalar/m;
```

在这个例子中，^符号匹配字符串的起始或新的一行的起始；$符号匹配任意行的末尾。上述语句都可匹配，因为输入被当作多行输入。如果删除模式中的 m，则输入就不匹配了。

### 7.4.4　单次匹配

每次对正则表达式求值时，Perl 5 都要花时间来解释它。如果模式包含变量且该变量在循环中使用，则每次循环时，Perl 5 都要解释这个模式。如果每时循环时模式中的变量不变，这样做的效率就会很低。在正则表达式中加入只匹配一次修饰符“o”则可以使程序更为简洁，运行速度更快，用法如下：

```
/pattern/o
```

只匹配一次修饰符在 Perl 5 解释器首次遇到正则表达时编译这个正则表达式，以后再遇到将被忽略，即整个程序中只解释一次。如果正则表达式在子程序中，则它只在第一次调用子程序时被解释，此后不再解释。以后又调用这个子程序时，使用第一次调用子程序时确定的模式值。

下面是一个例子：

**程序清单 7-11：忽略大小写的匹配**

```
$var = 1;
$line = <STDIN>;
while ($var < 10) {
    $result = $line =~ /$var/o;
    $line = <STDIN>;
    $var++;
}
```

该程序中，因在正则表达式中使用了只匹配一次修饰符“o”，它每次均匹配 /1/，因为后来的都是调用第一次的匹配值。

### 7.4.5 多余空格

多余空格修饰符“x”对正则表达式的模式匹配功能没有影响，它只是在正则表达式中插入多余空格。

读者可能已经体会到了，正则表达式难以阅读和理解，这时可以利用多余空格修饰符在正则表达式中加进说明。还可以在正则表达式中使用说明符“#”，使用正则和代码中使用说明语句一样，表示说明语句后面的所有内容均被忽略。

例如，可以将下面代码中的正则表达式：

```
if ($DED[$i] =~ /^$element\s+(\".*\")\s+(\w+)\s+(\d+)\s+(\d+)/ )
```

加上说明符，变成可读性更强的正则表达式：

```
if ($DED[$i] =~ /^$element\s+
(\".*\")\s+     # element description
(\w+)\s+        # element type
(\d+)\s+        # minimum value
(\d+)           # maximum value
(/x )
```

如果空格对模式不重要且模式不易看懂，则可以用多余空格修饰符分开模式原子。进行模式匹配操作时，正则表达式模式中的所有空格都将被忽略。如果正则表达式需要空格，则可以通过转义元字符或列表元字符加入，如：

```
/This[ ]is[ ]it/x
```

字符之间的空格忽略，每个 s 后面的空格是必需的，所以使用了列表元字符将其加入到正则表达式中。

## 7.5 与正则表达式相关的其他问题

### 7.5.1 模式匹配变量

每个正则表达式有三个特殊变量，这三个特殊变量提供了模式匹配结果的详细信息：

- 正则表达式匹配的字符放在特殊变量$&中。
- 将正则表达式匹配的字符前面的字符放在特殊变量$`中。
- 将正则表达式匹配的字符后面的字符放在特殊变量$'中。

Perl 5 解释器只在程序中使用了这些特殊变量时才对其赋值，也就是说，这些与变量相关的开销只在使用这些变量时才发生。一旦在程序中任何地方使用了任意模式匹配变量，即使此后不再使用，Perl 5 解释器也会一直对其赋值。这样，一旦用一个模式匹配了变量，就可以在代码中继续使用，而不会产生更大的开销。

下面是一个例子：

```
$_ = "Welcome to Great Wall!";
/G.*at/;      # 和 "Great"匹配
```

执行该语句后，$&现在是“Great”，$`是“Welcome to ”，$'是“ Wall!”。这些变量会在下次匹配时更改其值，若用户要在稍候用到这些值，必须另存起来。

下面是另一个例子：

**程序清单 7-12：使用模式匹配变量**

```
$string = "And little lambs eat ivy";
$string =~ /l[^s]*s/;
print "($`) ($&) ($')\n";
```

程序输出如下：

```
(And ) (little lambs) ( eat ivy)
```



代码中很少同时使用匹配前（$`）和匹配后（$'）变量。

### 7.5.2 替代函数

替换函数（s///）搜索字符串中的输入模式并将它转换成替换模式。其语法为：

```
s/pattern/replacement/
```

运算结果是将字符串中与pattern匹配的部分替换成replacement。

替换函数用于正则表达式，也符合正则表达式的一般正则。一般情况下，第一个出现的匹配模式被换成新模式，如果函数不约束替换函数，则函数修改缺省特殊变量$_。

下面是一个例子：

**程序清单 7-13：使用替代函数**

```
$needToReplace    = "bbb";
$replacementText = "1234567890";
$_ = "AAA bbb AAA";
$result = s/$needToReplace/$replacementText/;
print $result;
```

程序执行结果为：

```
AAA 1234567890 AAA
```

替代函数可以用 =~ 改变作用的目标，选择的目标必须可设给标量值，如标量变量或列表中的一个元素。下面是一个使用的例子：

```
$string = "This is a pen";
$string =~ s/pen/book/;       # 现在 $string = "This is a book"
```

替代函数后面可以使用本章第4节中介绍的六个模式修饰符，除此以外，替换字符串还可以使用“e”——作为表达式，它把替换部分的字符串看作表达式，在替换之前先计算其值，如：

```
$string = "0abc1";
$string =~ s/[a-zA-Z]+/$& x 2/e; # 现在 $string = "0abcabc1"
```

如果你想让替代函数在每个匹配的地方起作用，而非仅作用在第一个匹配处，可以在后

面加上一个参数“g”，如下：

```
$_ = "This is a piece of biscuit";
s/is/RE/g;          # 现在 $_ = "ThRE RE a piece of bREcuit"
```

替代函数后面加上参数“i”，表示忽略大小写。如果替代函数后面已经加了参数“g”，i出现在g的前面或后面部可以。

此外，如果你认为斜线不方面，也可以用他符号代替，只要三次都用同一个字符即可，如下所示：

```
s#is#RE#;
```

下面给出一个运用替代函数的比较复杂的例子：

**程序清单 7-14：使用替代函数的一个较复杂例子**

```
$_ = "I have 4 + 19 dollars and 8/2 cents.\n";
s{ (   \d+ \s*       # 匹配数字
       [+*/-]        # 匹配算术操作符
       \s* \d+       # 匹配其他数字
   )
}{ $1 }eegx;
print;
```

程序输出为：

```
I have 23 dollars and 4 cents.
```

## 7.5.3　内插变量

正则表达式内也可以使用内插变量，或者说，我们可以用内插变量“建造”正则表达式。请看下面的例子：

**程序清单 7-15：内插变量的使用**

```
$word = "dog";
$sentence = "Every dog has his day.";
if ($sentence =~ / \b$word\b/){
    print "There is a word $word in the sentence .\n ";
```

```
}
```

如果想把某些字符当成普通字符，而不是特殊符号，要在它们前面加上反斜线。有时候，这会有点困难，不过可以用\E 和\Q 符号来解决。请看下面的例子：

**程序清单 7-16：\E 和\Q 与内插变量一起使用**

```
$word = "[box]";
foreach   (qw(in[box] out[box] whit[box])){
    if (/ \Q$word\E/ ){
        print "$_ matched!\n";
    }
}
```

\Q 和\E 在这里会建构出\[box]\这样的正则表达式，而不会把整个字符串当成一个字符类别了。

## 7.5.4 模式次序变量

在模式匹配后调用重用部分的结果可用变量$n，全部的结果用变量$&，如下例所示：

```
$string = "This string contains the number 325.467.";
$string =~ /-?(\d+)\.?(\d+)/;     # 匹配结果为325.467
$integerpart = $1;    #   现在 $integerpart = 325
$decimalpart = $2;     #   现在 $decimalpart = 467
$totalpart = $&;      #   现在 $totalpart = 325.467
```

在替换部分可使用模式次序变量$n，但在替换部分不支持模式的特殊字符，如{}，*，+等，如 s/abc/[def]/将把 abc 替换为[def]。

下面的例子将会帮助你更好地理解模式次序变量：

**程序清单 7-17：模式次序变量的使用**

```
$x = 'Perl is free.';
$y = 'ManagerWare costs $99.95.';
foreach ($x, $y) {
    /^(\w+) (?:is|(costs)) (?(2)(\$\d+)|\w+)/;   # Either (\$\d+) or \w+
    if ($3) {
        print "$1 costs money.\n";              # ManagerWare costs money.
    } else {
        print "$1 doesn't cost money.\n";   # Perl doesn't cost money.
```

```
    }
}
```

程序输出结果为：

```
Perl doesn't cost money.
ManagerWare costs money.
```

### 7.5.5 扩展模式匹配

Perl 还支持 PERL4 和标准 VINX 模式匹配操作所没有的一些模式匹配能力。其语法为：(?<c>pattern)，其中 c 是一个字符，pattern 是起作用的模式或子模式。

1. 不存储括号内的匹配内容

在 Perl 的模式中，括号内的子模式将被存储在内存中，扩展模式功能：需要否取消存储该括号内的匹配内容，如：

```
/(?:a|b|c)(d|e)f\1/;
```

其中：1 表示已匹配的 d 或 e，而不是 a 或 b 或 c。

2. 内嵌模式选项

通常模式修饰符置于其后，有四个选项：i、m、s、x（这些选项的意义前面已经介绍过）可以内嵌使用，它的使用语法如下：

```
/(?option)pattern/;
```

等价于：

```
/pattern/option;
```

3. 肯定的和否定的预见匹配

肯定的预见匹配语法为：

```
/pattern(?=string)/;
```

其意义为匹配后面为 string 的模式；相反，(?!string)的意义为匹配后面非 string 的模式，下面是一个例子：

```
$string = "25abc8";
$string =~ /abc(?=[0-9])/;
$matched = $&;     # $&为已匹配的模式，此处为abc，而不是abc8
```

## 7.6 小结与练习

### 7.6.1 小结

本章主要介绍了 Perl 中的正则表达式。利用正则表达式作为编程工具，可以编写出强大的代码。正则表达式可以说是最能体现 Perl 优越性的功能之一。虽然乍一看起来，正则表达式很难学，但只要细心去体会，你也能掌握它。本章的目的是揭开正则表达式神秘的面纱，使你对它不再望而却步。

本章第一节给出了正则表达式的定义，并通过一个简单的例子直观地说明了什么是正则表达式 。

第二节介绍了正则表达式的构成。元字符是系统定义的一些特殊字符，原子构成了正则表达式的基本单元。简言之，原子就是模式中的一个单个项目。元字符不是原子，它要与原子配对时才有意义。

第三节介绍了正则表达式的正则。正则表达式有单字符型、组合型和定位类型，各种功能强大的正则表达式都是由它们组成的。然后介绍了各种类型操作符的优先级，即谁先谁后的问题。最后对每种类型的正则表达式都给出了具体的例子，以帮助读者理解它们。

第四节介绍了正则表达式的模式修饰符。模式修饰符改变了整个正则表达式的操作。这种模式修饰符包括全局匹配符“g”、忽略大小写匹配符“I”、多行匹配符“m”、单次匹配符“o”和多余空格修饰符“x”。熟练运用它们是掌握正则表达式的关键。

本章最后一节介绍了与正则表达式相关的一些其他问题。每个正则表达式都有三个特殊变量，分别表示匹配字符串之前的字符串、匹配的字符串和匹配字符串后面的字符串。替换函数（s///）搜索字符串中的输入模式并将它换成替换模式。正则表达式内也可以使用内插变量，因此我们可以用内插变量构造正则表达式，这使正则表达式更加灵活。

模式次序变量的目的在于分别获得多个匹配中的每一项。另外本节还介绍了几个扩展模式匹配，它们一般使用很少。

学习了本章，读者应该对 Perl 中的正则表达式不再陌生了，但要真正掌握正则表达式的应用还要靠多实践，所以应该尽量在代码中使用正则表达式。记住，循序渐进，由易到难，这样可以建立信心。

下一章将介绍字符串操作，它是正则表达式的一个很重要的应用。

### 7.6.2 习题与思考

（1）写出下面程序的输出结果。

```
print "Please input your operating system : \n";
$_ = chomp(<STDIN>);
if (/Windows/) {
   s/Windows/Linux/;
}
print "Now you operating system is: $_\n";
```

（2）构造一个正则表达式，使它能够匹配：

① 至少一个 a，后跟任何数量的 b。

② 任意个数的反斜杠，后跟任意数量的*号。

③ 连续三个在$whatever 里面的值。

④ 任意五个字符，包含换行符。

（3）写一个程序，从键盘输入若干行单词，从中找出包含所有五个元音字母（a, e, i, o 和 u）的行，并输出。

（4）写出下列正则表达式分别匹配哪些字符串。

```
/com(poundl)/;
/com(pound(sl)l)/;
/com(pound)?/;
/com(pound(s?))?/;
/com(pounds?)?/;
```

（5）写出下面程序的输出结果。

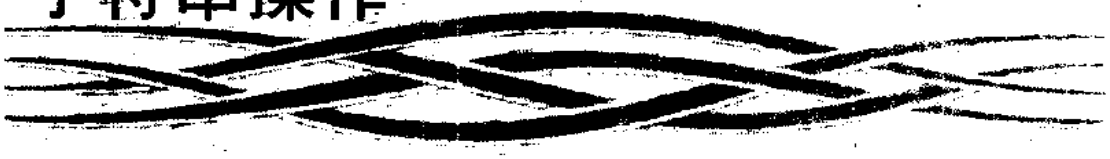
```
($rows, $cols) = (24, 80);
$text = q(I am $rows high and $cols long);   # like single quotes!
$text =~ s/\$(\w+)/${$1}/g;
print $text,"\n";
$text = "thIS is a loNG liNE";
$text =~ s/(\w+)/\u\L$1/g;
print $text;
```

（6）写出下面程序的输出结果。

```
$_ = "The year 1752 lost 10 days on the 3rd of September";
while (/(\d+)/gc) {
    print "Found number $1\n";
}
if (/\G(\S+)/g) {
    print "Found $1 after the last number.\n";
}
```

# 第8章

# 字符串操作



字符串和数值的相互转换
把标量插入字符串
字符串处理函数
小结与练习



Perl 5 的灵魂就在于字符串操作。Perl 是从操作字符串开始发展的，目前的普及仍然与字符串有关。对字符串的操作很丰富，本章介绍的函数可以处理字符串中的字符，把字符串变成数值，修改字符串的大小写，改变字符串中的某些字符，从字符串中取出子字符串，分离与连接字符串中的某个字符等。字符串的操作是与正则表达式紧密联系的，学习了本章，读者将会对正则表达式的应用有更深的了解，也会明白 Perl 为什么以字符串的处理见长。

本章要点：

- 数字与字符和字符串的相互转换以及它们之间的区别。
- 在字符串中插入标量。
- 字符串处理函数。

# 8.1 字符串和数值的相互转换

## 8.1.1 数字与字符的相互转换

Perl 5 提供有函数可对数字与字符进行相互转换。

### 1. 数字变成字符

可以利用 chr 函数将数字变成字符。

程序读取键盘输入并使用显示缓存时，用这个函数将非打印字符输送到屏幕上。chr 函数的用法如下：

```
lvalue=chr (number);
```

数字可以是任何有效的 ASCII 数。chr 函数通常使用八进制和十六进制值，因为它们与定义屏幕和控制符的字节代码关系更密切。

下面是一个例子：

```
$char=chr(65);      # 现在 $char=A
```

### 2. 字符变成数字

ord 函数可将 ASCII 字符变成数字值，其作用与 chr 函数相反。

ord 函数的用法如下：

```
lvalue=ord (expression);
```

ord 函数只将表达式中的第一个字符变成数字值。如果省略表达式，则 ord 函数使用缺省特殊变量$_。

下面是数字和字符转化的一个例子：

**程序清单 8-1：字符和数字的相互转化**

```
%associativeArray=("Jack A.", "Dec 2","Joe B.",
 "June 2", "Jane C.", "Feb 1
$number=ord ("apple");
$ascii_value=ord("e");
$character=chr(101);
$num=ord($char);
$char=chr($num);
print "The number of apple is : $number\n";
print "The number of e is : $ascii_value\n";
print "The character of 101 is : $character\n";
```

程序执行结果为：

```
The number of apple is : 97
The number of e is:101
The character of 101 is : e
```

### 8.1.2 数字与字符串的相互转换

在 Perl 中，字符串和数值也可以相互转换。注意，这里是“**字符串**”而非“**字符**”，这与上面是完全不同的。请看下面的例子：

```
$string="43";
$number=28;
$result=$string + $number;      # $result=71
```

现在大家应该明白了吧，8.1.1 节讨论的用函数转换的是同一个字符的不同表现形式（ASCII 码形式和字符形式），实际上仍然是相同的内容；而这里是将字符串与数字作直接的转化，形式上看来好像没变，但在内容上已经完全变了。

若字符串中含有非数字的字符，则从左起至第一个非数字的字符，如：

```
$result="hello" * 5;      #$result=0
```

```
$result="12a34" +1;      #$result=13
```

## 8.2 把标量插入字符串

如果字符串用“双引号表示法”表示，我们就可以在其中插入变量。当程序进行到该处时，字符串的变量部分就会用该变量当时的值取代。如果变量当时并未赋值，系统会用空字符串替代该变量所在的位置。请看下面的例子：

```
$x="news" ;
$y="$x report";          #     $y的值变为 "news report"
```

在这里，替代动作只会进行一次，意思是，取代值若内含“$”号，系统并不会把“$”号后面的字符串看成另外一个变量，再做一次替代动作。例如：

```
$x="$news" ;     # $x的内容中本来就有一个美元符号
$y="$x report";     # $y的值变为 "$news report"
```

当不希望系统进行替代动作的时候，可以把会引起歧义的部分用“单引号表示法”表示、或在“$”号前面加反斜线，让“$”号的特殊意义失效：

```
$x="$news" ;
$y1="report " . '$x';          #   $y1的值现在为 "report $x"
$y2="report \$x";          #   与上面一样，$y2的值现在为 "report $x"
```

Perl 进行置换时，会尽可能寻找最长的变量名，这一点必须注意。为避免混淆，可以在变量的名字前后用大括号括起来，或用字符串串接的方法来限定匹配的变量。下面是一个例子：

**程序清单 8-2：把标量插入字符串**

```
$passer="news" ;
$passerby="report";
$y1="This is a piece of $passerby";     # $y1的值现在为 "This is a piece of report"
$y2="This is a piece of ${passer}by";     # $y2的值现在为 "This is a piece of newsby"
$y3='This is a piece of $passer" . "by";     # $y3的值现在为 "This is a piece of newsby"
print $y1,"\n";
print $y2,"\n";
```

```
print $y3,"\n";
```

程序执行结果为：

```
This is a piece of report
This is a piece of newsby
This is a piece of newsby
```

## 8.3　字符串处理函数

表 8-1 是字符串处理函数的列表，表中给出了各个字符串处理函数的调用语法和返回值，并给出了说明。

表 8-1　　字符串处理函数

| 函数名 | 调用语法 | 返回值 | 说明 |
| --- | --- | --- | --- |
| index | positio=index (string, substring, position); | 返回子字符串 substring 在字符串 string 中的位置，如果不存在则返回-1 | 参数 position 是可选项，表示匹配之前跳过的字符数，也就是说返回值必须大于等于这个数 |
| rindex | position=rindex(string, substring, position); | 返回子串 substring 在字符串 string 中的位置，如果不存在则返回-1 | 与 index 类似，区别是从右端开始匹配。参数 position 是可选项，表示匹配之前跳过的字符数，也就是说返回值必须小于等于这个数 |
| length | num=length (string); | 返回字符串长度 | |
| pos | offset=pos(string); | 返回最后一次匹配的位置 | |
| substr | substr (expr, skipchars, length) | 若长度为 0 或开始位置大于字符串长度，则返回空字符串；否则返回 expr 中从 skipchars 位置开始的 length 个字符组成的子串 | 第一个字符位置为 0；子串长度参数 length 可忽略，意味着取剩下的全部字符；当此函数出现在等式左边时，expr 必须为变量或数组元素，此时其中部分子串被等式右边的值替换 |
| study | study (scalar); | 无 | 用一种内部格式提高变量的访问速度，同一时刻只对一个变量起作用 |
| lc | retval=lc(string); | 转换之后的字符串 | 将字符串全部转换成小写字母 |

续表

| 函数名 | 调用语法 | 返回值 | 说明 |
| --- | --- | --- | --- |
| uc | retval=uc(string); | 转换之后的字符串 | 将字符串全部转换成大写字母 |
| lcfirst | retval=lcfirst(string); | 转换之后的字符串 | 将第一个字母转换成小写 |
| ucfirst | retval=ucfirst(string); | 转换之后的字符串 | 将第一个字母转换成大写 |
| quotameta | newstring=quotemeta ( oldstring); | 转换之后的字符串 | 在非单词的字母前面加上反斜线（\）常用于模式匹配操作中，确保字符串中没有字符被看作匹配操作符 |
| Join | join (joinstr, list); | 连接之后的字符串 | 把字符串列表(数组)组合成一个长的字符串，在每两个列表元素间插入joinstr |
| sprintf | sprintf (string, fields); | | 输出的结果作为返回值赋给变量 |

### 8.3.1 改变大小写函数

我们知道，Perl 5 是对大小写敏感的，所以在编写程序或别的内容时，区分大小写是很重要的。

Perl 5 有四个用以改变大小写的函数：lc、lcfirst、uc 和 ucfirst。

它们的语法如下：

```
lvalue=characterCaseFunction (expression);
```

表达式是可选的，如果省略，则使用缺省特殊变量$_。

函数将整个表达式或第一个字符变为对应的大写或小写字母，这四个函数的功能分别如下：

lc 函数——将表达式中所有字符变成小写

lcfirst 函数——只将表达式中第一个字符变成小写

uc 函数——将表达式中所有字符变成大写

ucfirst 函数——只将表达式中第一个字符变成大写

如果字符不是 A 到 Z 或 a 到 z 的 ASCII 字符，则不受改变大小写函数的影响。

下面是两个例子：

**程序清单 8-3：改变大小写函数的应用一**

```
$letter="Hello World!";
$transform=lc($letter);
print "The result of \"lc\" to \"Hello World!\" is : $transform\n ";
```

```
$transform=lcfirst($letter);
print "The result of \"lcfirst\" to \"Hello World!\" is : $transform\n ";
$transform=uc($letter);
print "The result of \"uc\" to \"Hello World!\" is : $transform\n ";
$transform=ucfirst($letter);
print "The result of \"ucfirst\" to \"Hello World!\" is : $transform\n ";
```

程序执行结果为：

```
The result of "lc" to "Hello World!" is : hello world!
The result of "lcfirst" to "Hello World!" is : hello World!
The result of "uc" to "Hello World!" is : HELLO WORLD!
The result of "ucfirst" to "Hello World!" is : Hello World!
```

大家可以通过仔细分析这些结果来体会一下改变大小写函数的作用。从上面的例子可以看出，改变大小写函数也可以针对变量，下面是另一个例子：

**程序清单 8-4：改变大小写函数的应用二**

```
$little="John Smith";
$big=uc($little);
print "Big: $big\n";
$little=lc($big);
print "little: $big\n";
print ucfirst($little),"\n";
print lcfirst($little);
```

程序输出结果为：

```
Big: JOHN SMITH
little: JOHN SMITH
John smith
john smith
```

## 8.3.2　字符串截尾函数

1. chop 函数

chop 是一个很有用的函数，我们在前面已经见过它了，这里再详细介绍一下。它以一个

标量变量做为参数，并删除所传入的参数的最后一个字符。例如：

```
$x="hello world";
chop ($x);      # $x现在是  "hello worl"
```

这里参数的值会被改变，所以不能用标量数值作为该函数的参数。注意，它返回的是被删除的字符。请看下面的例子：

```
$x="hello world";
$x=chop ($x);      # $x现在是"d"
```

如果给 chop 空字符串，它不会做任何事，也不会返回任何值，系统甚至不会发出任何错误信息。在 Perl 里，大部分动作都没有“边界限制”，意思是你可以进行到最后面(或最前面)，系统也不会抱怨一句。某些人声称这是 Perl“结构上的缺点”，不过也有许多高手写程序写得很高兴。字符串变量还可用 chop 函数连续“截尾”，见下面的例子：

```
$x="good morning\n";
chop   $x;    # $x现在是   "good morning"
chop   $x;    # $x现在是   "good mornin"
```

2. chomp 函数

如果读者不能确定字符串最后面是否有换行符号，可以改用比较安全的 chomp 函数。该函数只会删除换行字符，见下面：

```
$x="good morning\n";
chomp   $x;    # $x现在是   "good morning"
chomp   $x;     # $x的值并未改变，仍然是   "good morning"
```

下面是一个具体的例子：

**程序清单 8-5：使用 chomp 函数**

```
$words=<STDIN>;
chomp($words);
while($words) {
        print $words;
        $words=<STDIN>;
}
```

程序输出结果为：

```
ok
okyes
yes
all   right!
all   right!
```

之所以在输入第一个单词“ok”之后没有回车，就是因为回车已被 chomp 函数截掉了。而后来的循环中接受的输入均带有回车，就是因为循环中并没有使用 chomp 函数。

现在将程序变一下：

**程序清单 8-6：chomp 函数的应用**

```
$words=<STDIN>;
while($words) {
        chomp($words);
        print $words;
        $words=<STDIN>;
}
```

程序执行结果为：

```
ok
okyes
yesall
all
```

### 8.3.3　搜寻子字符串函数

1. index 函数

搜寻子字符串有很多种情况，如果只是在大字符串中查找其中一部分，是非常简单的，使用 index 函数就可以了，其方法如下：

```
$x=index ($string , $substring);
```

Perl 会找出 string 中第一次出现 substring 的位置，并返回一个代表该位置的整数。此整数是从零开始算的，即：如果子字符串出现在字符串开始，index 会返回 0；如果从第二个字符开始出现，返回 1，依此类推。如果找不到，则返回-1。

下面是一些例子：

```
$where= index ("weight","e");       # $where 得到 1
$person="tom";
$where=index ("hello tom" , $person);      # $where 得到 6
$where=index ("hello tom" , "weight");     # $where 得到 -1
$x="good";
$where=index ("hello handy", chop( $x));     # $where 得到 9
```

子字符串若出现多次，index 会返回最左边的位置。若想找到较后面的位置，就必须加入第三个参数，代表返回值必须大于等于这个数。请看下面的用法：

```
$where=index ($string,$substring,$skip);
```

下面是几个例子：

```
$x=index ("how are you","o");       # 返回 1 (第一个o，即how 中的o)
$x=index ("how are you","o",0);      # 返回 1 ，同上
$x=index ("how are you","o",3);      # 返回 9 (第二个o，即you 中的o)
$x=index ("how are you","o",10);      # 返回 -1 ，因为第10个字符后面找不到o
```

2. rindex 函数

另外还有一个类似的函数称为 rindex，它会由字符串的右边开始搜寻子字符串，不过返回值的意义仍然和 index 一样。rIndex 也可以有第三个参数，代表返回值必须小于等于这个数。请看下面的例子：

```
$x=index ("how are you","a");       # 返回 4
$x=index ("how are you","o");       # 返回 9 (you 中的o)
$x=index ("how are you","o",3);      # 返回 1(how 中的o)
$x=index ("how are you","r",4);      # 返回 -1 ，因为第4个字符前面找不到r
```

### 8.3.4 子字符串的展开与替代

我们可以用一般语句替代字符串的内容，但如果只知道字符位置，这样做会很没效率。比较好的解决方法是使用 substr 函数。这个函数有三个参数，分别是字符串值、开始位置以及长度，其调用语法如下：

```
$s=substr ($string,$start,$length);
```

开始位置和 index 一样，第一个字符为第 0 位。该函数表示在字符串$string 中从位置$start 开始，取$length 个字符。它会在字符串结尾停止，所以就算开始位置加长度大于字符串总长度，也不会出错。请看下面的例子：

```
$x="how are you";
$grab=substr ($x,4,3);        # $grab 现在是"are"
$grab=substr ($x,8,1000);     # $grab 现在是"you"
```

如果要产生 10 的某次方，只要次数不是太高，就可以用下面这种方式得到：

```
$x=substr("10000000000",0,$power+1 );      # 10的$power次方
```

如果长度为 0，会返回空字符串；如果开始或结束位置小于 0，则被认为是从字符串最后面算起的这么多个的位置。请看下面的例子：

```
$x=substr("how are you",-3,3 );     # 最后三个字符，即"you"
$x=substr("how are you",-2,3 );     # 最后两个字符，即"ou"
$x=substr("how are you",-3,1 );     #后面起的第三个字符，即"y"
```

如果开始位置在字符串开头之前（如绝对值大于字符串真正长度的负数），系统会把字符串开始位置当成真正的开始位置，如同把第二个参数设成 0；若开始位置大于字符串真正的长度，会返回空字符串；若省略长度参数，就如同将那个参数设成无穷大的整数，所以会得到由开始位置到字符串结尾的所有内容。

下面的程序有利于理解 substr 函数的返回值：

**程序清单 8-7：substr 函数的应用**

```
$string="This is what you have";
$first =substr($string, 0, 1);     # "T"
print $first,"\n";
$start =substr($string, 5, 2);     # "is"
print $start,"\n";
$rest  =substr($string, 13);       # "you have"
print $rest,"\n";
$last  =substr($string, -1);       # "e"
print $last,"\n";
$end   =substr($string, -4);       # "have"
print $end,"\n";
```

```
$piece =substr($string, -8, 3);     # "you"
print $piece,"\n";
```

程序的输出结果为：

```
T
is
you have
e
have
you
```

如果 substr 的第一个参数是标量变量，substr 本身也可以出现在等号左边。这样的写法，可以将 substr 返回的某段子字符串替代为别的内容。下面是一个例子：

```
$x="how are you";
substr($x,0,3 )="where";      # $x现在是"where are you"
```

下面是运用 substr 函数的一个例子，读者可以从程序的输出中体会它的作用：

**程序清单 8-8：用字符串替代 substr 函数返回的内容**

```
$string="This is what you have";
print $string,"\n";
substr($string, 5, 2)="wasn't";     # change "is" to "wasn't"
print $string,"\n";
substr($string, -12) ="ondrous";# "This wasn't wondrous"
print $string,"\n";
substr($string, 0, 1)="";          # delete first character
print $string,"\n";
substr($string, -10) ="";          # delete last 10 characters
print $string,"\n";
```

程序输出为：

```
This is what you have
This wasn't what you have
This wasn't wondrous
his wasn't wondrous
his wasn'
```

从上面的例子中可以看出，要替代的字符串不必和被替代的字符串一样长，原字符串会正确地进行缩放。

### 8.3.5　字符串转换函数

拿到一个字符串，想把其中某些字符换成新字符，例如想把所有的 a 换成 b，以及所有的 b 换成 a，该怎么办呢？Perl 提供了一个操作符 tr 来解决这一问题，其用法如下：

```
tr /ab/ba/;
```

tr 操作符接受两个参数，分别是旧字符串和新字符串。tr 操作符每个参数之间，包括第二个参数后面，都必须加上分隔符号。注意分隔符号可以任意指定，我们这里使用的是斜线。tr 操作符会更改$_变量（关于变量$_后面将会讨论）的内容。在其中搜寻和第一个参数相同的字符串，所有找到的字符串会替被替换成第二个参数中对应位置上的字符。下面是一些例子：

**程序清单 8-9：字符串转换函数的使用**

```
$_="how are you";
$count=tr /ho/oh/;      # $_现在成为 "ohw are yhu"，$count为3
print " The string now is : $_，the character number of replace is :$count\n ";
tr /abcde/ABCDE/;       # $_现在成为 "ohw ARE yhu"
print " The string now is : $_,the character number of replace is :$count\n ";
tr /a-z/A-Z/;       # $_现在成为 "OHW ARE YHU"
print " The string now is : $_,the character number of replace is :$count\n ";
```

程序执行结果为：

```
The string now is : ohw are yhu,the character number of replace is :3
The string now is : ohw ArE yhu,the character number of replace is :3
The string now is : OHW ARE YHU,the character number of replace is :3
```

如果新字符串比原字符串短，替代时会重复新字符串中的最后一个字符，直到旧字符串的长度为止。

```
$_= "how are you";
tr /a-z/x/;      # $_现在成为 "xxx xxx xxx"
```

要避免这种情形，可以在 tr///表达式的最后面后面加 d 字符选项，意义是“delete”，如

此就不会一直重复最后一个字符。任何符合条件的旧字符串字符，如果新字符串对应处没有字符，则被删除，像这样：

```
$_= "how are you";
tr /a-z/ABCDE/d;     # $_现在成为 " ARE "
```

如果在操作符后面加 c 选项，代表“第一个参数是反过来写的”。意思是，第一个参数其实应该是“所有 256 个 ASCII 字符，扣掉我们在第一个参数位置上写的那些字符”。请看下面的例子：

```
$_= "how are you";
tr /a-z/_/c;     # $_现在成为 "how_are_you"
tr /a-z//cd;     # $_现在成为 "howareyou"
```

注意，在此 c 和 d 两个选项可以同时使用。在上面第二个例子中，我们先对第一个参数“取补数”，所以它的意思应该是“所有非字母字符”，最后再用 d 选项删除这些字符。

最后要介绍的选项是 s。如果被替代的部分出现连续重复的字符，它会把这部分“缩”成一个。请看下面的例子：

```
$_= "111222abcdef";
```

在这里 abc 被替换成 567，如果没有 S 选项的话 def 本来应该是 xxx，但由于后面使用了 S 选项，因此现在只剩下一个 x。另外，字符串前面的“111222”没被缩水，因为它们不是被替代的部分。

### 8.3.6 取字符串长度函数

与在 C 语言中所用到的相似，Perl 5 对字符串的操作也提供有 length 函数。其语法如下：

```
lvalue=length (expression);
```

该函数读取字符串的内容，并返回字符串中的字符个数。如果变量没有声明，函数就会返回 0。

下面是一个例子：

```
$string="Hello!";
$number=length($string);
```

该段代码执行后，$number=6。

下面是另一个例子：

```
$string="How are you doing!";
$matchedString =~ /Are/ ;
$number=length($&);
```

该段代码执行后，$number=0。因为匹配到的字符串为空，其长度当然为 0。

### 8.3.7　分离和连接字符串函数

正则表达式可以用来把一个字符串分成多个字段，split 函数可以做到这件事，而 join 函数则可以把这些字段组合起来。

1. split 函数

split 函数有两个参数，分别是正则表达式和字符串，它会寻找字符串中匹配正则表达式的部分，其他不匹配的部分会依序用列表值的方式传回。

请看下面的例子：

```
$line="ok::15:32:hello:/yes/no:/user/bin/perl";
@field=split (/:/,$line);
```

现在 @field 的值为（“ok”,“,” “15”，“32”，“hello”，“/yes/no”，“/user/bin/perl”）。注意，第二个元素的值为空字符串。如果不想要这样的结果，可以用所有的冒号做分隔符：

```
$line="ok::15:32:hello:/yes/no:/user/bin/perl";
@field=split (/:+/,$line);
```

这样 split 函数里面的正则表达式就会和一个或一个以上的冒号相匹配，从而就不会出现空字符串了。

下面是一个使用 split 函数的例子：

**程序清单 8-10：字符串分离函数的使用**

```
$string="an apple a day";
print $string,"\n";
foreach $byte (split / /, $string) {
    print $byte,"\n";;
}
```

程序输出结果为：

```
an apple a day
an
apple
a
day
```

split 函数的第一个参数即正则表达式中，还可以有多个分割符，表示在碰到它们之一时进行字符分割，下面就是一个例子：

**程序清单 8-11：在 split 函数中使用多个分割符**

```
$string= "1-10, 20";
print $string,"\n";
foreach $byte (split /([-, ])/, $string) {
    print $byte,"\n";;
}
```

程序输出为：

```
1-10,20
1
-
10
,
20
```



$_是 split 函数第二个参数的默认值。

2. join 函数

分离容易，连接则难。但 Perl 5 提供的 join 函数将列表变成一个分割字符很容易。join 函数的语法如下：

```
lvalue=join (delimiter,list);
```

左边值应为标量变量，分割符可以是任何表达式。该函数的第二个参数是列表值，它会用第一个参数字符串当“胶”，把第二个参数里面的值一个个“粘”起来。

下面是一个例子：

```
@list=("This", "is", "a", "book" );
$line=join ("_",@list);
```

执行后：
$line="This_is_a_book"。

下面是另外一个例子：

**程序清单 8-12：字符串连接函数的使用**

```
@list=(1, '-', undef, 10, undef, ',', 20);
print @list,"\n";
$string=join('**',@list);
print $string;
```

程序执行结果为：

```
1-10,20
1**-****10****,**20
```

请注意输出的结果。它说明 undef 在输出时并没有什么体现，就像没有它似的，但它确实在数组中存在着，占据着一项，这从 join 函数的输出结果可以看出来。

## 8.4　小结与练习

### 8.4.1　小结

因为对字符串的强大处理功能是 Perl 的优势之一，所以本章专门介绍了对字符串的常用操作，其中大量利用了上一章中正则表达式的内容。可以说正是有了正则表达式，才使得 Perl 对字符串的操作变得十分容易。

本章第 1 节给出了数字和字符的互相转换方法。可以利用 chr 函数将数字变成字符；利用 ord 函数将 ASCII 字符变成数字值，其作用与 chr 函数相反。

第 2 节介绍了如何将标量插入字符串。当程序进行到变量处时，字符串的变量部分就会用该变量当时的值取代。这使对字符串的处理变得十分灵活。

第 3 节介绍了字符串处理函数。改变大小写函数可以改变字符串中一定位置上字符的大小写；字符串截尾函数 chom 可以删除掉字符串尾部的字符；chomp 函数仅仅去掉字符串末

尾的换行符，如果没有换行符则不起作用；index 函数从左边开始搜索指定字符串中某字符串第一次出现的位置；而 rindex 函数从右边开始搜索指定字符串中某字符串第一次出现的位置；字符串替代函数 substr 可以替代给定字符串中从某位置开始的指定长度的字符；字符串转换函数 tr 可以将匹配模式的字符串转换成指定的字符串；length 函数可以获取字符串的长度；split 函数可以将字符串分割成若干个字符串；而 join 函数则可以将若干个字符串合并成一个字符串。

学习了本章，读者应该掌握用 Perl 中的正则表达式来处理字符串的基本方法，下一章将重点介绍一些自定义函数。

### 8.4.2 习题与思考

（1）写出下面程序的输出结果。

```
$beast  ="dromedary";
$capit  =ucfirst($beast);
print $capit,"\n";
$capit  ="\u\L$beast";
print $capit,"\n";
$capall =uc($beast);
print $capall,"\n";
$capall ="\U$beast";
print $capall,"\n";
$caprest=lcfirst(uc($beast));
print $caprest,"\n";
$caprest="\l\U$beast";
print $capress,"\n";
```

（2）指出下面各个语句的作用效果。

```
tr/aeiou/!/;
tr{/\\\r\n\b\f. }{_};
tr/A-Z/a-z/ ;
$count=tr/0-9//;
tr/@$%*//d;
```

（3）写出下面程序的输出结果：

```
$string="I love you!";
$lookfor="love";
$pos=-1;
```

```
while (($pos=index($string, $lookfor, $pos)) > -1) {
    print "Found $lookfor at $pos in \" $string.\"\n";
    $pos++;
}
$pos=length $string;
while (($pos=rindex $string, $lookfor, $pos) >= 0) {
    print "Found $lookfor at $pos in \" $string.\"\n";
    $pos--;
}
```

（4）写出下面程序的输出结果。

```
$answer=substr <STDIN>, 0, -1;
print $answer,"\n";
$var="I am OK!";
print "The string before replace :$var\n";
$last_char=chop($var);
$last_char=substr($var, -1, 1, '');    # same thing
print "After replaced: ",$var;
print "\nThe return is $last_char";
$caravan="Thank you!";
print "\nBefore replace: $caravan\n";
substr($caravan, -5)="";
print "After replaced : $caravan\n";
```

# 第9章

# 自定义函数

一些常用的系统函数及其使用
自定义函数的定义
函数的调用
更方便地使用函数
Perl 5 中预定义的函数
函数的应用举例
小结与练习



到目前为止，我们所介绍的函数都是 Perl 5 自带的内置函数，如 print、chop 等。虽然它们都是很通用的，但仅有这些通用的函数还是不够的，当读者要处理一些特殊问题时，它就显得无能为力了，或十分的繁琐。所以有必要定义自己的函数。现在我们就研究一下如何定义自己的函数。

除了函数之外，还有子程序，那么它们是什么关系呢？子程序就是可重复使用的代码块，而函数是返回数值的可复用代码块。所有 Perl 5 子程序都返回最后求值表达式的值，因此在 Perl 5 中子程序和函数是同一个意思，下面将按照习惯称之为函数。

本章要点：

- 自定义函数的定义及调用。
- 在函数中使用参数。
- 在函数中使用私有变量和半私有变量。

## 9.1 一些常用的系统函数及其使用

表 9-1 列出了常用的系统函数，表 9-2 则列出了 pack 函数的格式字符。**表 9-1 常用的系统函数列表**

| 函数名 | 调用语法 | 返回值 | 说明续表 |
|---|---|---|---|
| eval | eval(string) | 无 | 将 string 看作 Perl 语句执行。正确执行后，系统变量$@为空串，如果有错误，$@中为错误信息 |
| system | system(list) | 返回值由程序结束后的错误代码左移八位而成 | list 中第一个元素为程序名，其余为参数。system 启动一个进程运行程序并等待其结束 |
| fork | procid = fork(); | 子进程返回零，父进程返回非零值，此值为子程序的进程 ID 号 | 创建程序的两个拷贝：父进程和子进程，并同时运行 |
| pipe | pipe (infile, outfile); | 无 | 与 fork 合用，给父进程和子进程提供通信的方式。送到 outfile 文件变量的信息可以通过 infile 文件变量读取 |
| exec | exec (list); | 无 | 与 system 类似，区别是启动新进程前结束当前程序。常与 fork 合用，当 fork 分成两个进程后，子进程使用 exec 启动另一个程序 |

续表

| 函数名 | 调用语法 | 返回值 | 说明 |
| --- | --- | --- | --- |
| syscall | syscall (list); | 无 | 调用系统函数，list 第一个元素是系统调用名，其余为参数。如果参数是数字，就转化成 C 的整型数；否则传递字符串指针 |
| die | die (message); | 无 | 终止程序并向 STDERR 输出错误信息。message 可以为字符串或列表。如果最后一个参数不包含换行符，则程序文件名和行号也被输出 |
| warn | warn (message); | 无 | 与 die 类似，区别是不终止程序 |
| exit | exit (retcode); | 指定的返回值 | 终止程序并指定返回值 |
| kill | kill (signal, proclist); | 无 | 给一组进程发送信号。signal 是发送的数字信号，9 为杀掉进程。proclist 是进程 ID 列表 |
| sleep | sleep (time); | 返回值为实际停止的秒数 | 将程序暂停一段时间，time 为暂停秒数 |
| wait | procid = wait(); | 返回值为子进程 ID，如果没有子进程，则返回-1 | 暂停程序执行，等待子进程终止，不需要参数 |
| waitpid | waitpid(procid, waitflag); | 无 | 暂停程序执行，等待特定的子进程终止。Procid 为等待的进程 ID |
| caller | subinfo = caller(); | 返回调用者的程序名和行号，用于 Perl Debugger | 返回值为三元素的列表：即<br>调用处的包名、调用者文件名、调用处的行号 |
| chroot | chroot (dir); | 无 | 改变程序的根目录 |
| Local | local($variable); | 无 | 在语句块中定义局域变量，仅在此语句块中起作用，对其的改变不对块外同名变量造成影响 |
| times | timelist = times | 返回该程序及所有子进程消耗的工作时间 | 返回值为四个浮点数的列表：即<br>程序耗用的用户时间、程序耗用的系统时间、子进程耗用的用户时间、子进程耗用的系统时间 |

表 9-2 是用于 pack 函数的完整的格式字符：

表 9-2　　pack 函数的格式字符：

| 格式字符 | 描　述 |
|---|---|
| a | 用空字符(null)补足的字符串 |
| A | 用空格补足的字符串 |
| b | 位串，低位在前 |
| B | 位串，高位在前 |
| c | 带符号字符（通常为 -128~127） |
| C | 无符号字符（通常为 8 位） |
| d | 双精度浮点数 |
| f | 单精度浮点数 |
| h | 十六进制数串，低位在前 |
| H | 十六进制数串，高位在前 |
| i | 带符号整数 |
| I | 无符号整数 |
| l | 带符号长整数 |
| L | 无符号长整数 |
| n | 网络序短整数 |
| N | 网络序长整数 |
| p | 字符串指针 |
| s | 带符号短整数 |
| S | 无符号短整数 |
| u | 转化成 uuencode 格式 |
| v | VAX 序短整数 |
| V | VAX 序长整数 |
| x | 一个空字节 |
| X | 回退一个字节 |
| @ | 以空字节(null)填充 |

## 9.2 自定义函数的定义

在 Perl 5 中，自定义函数用关键字 sub 声明或定义。子程序可以命名或匿名，声明的结构如下：

```
sub subname ( parameterList){
    statement1;
    statement2;
    statement3;
    ... ...
}
```

其中 subname 代表自定义函数的名称，它的命名规则和标量变量、散列、数组一样。函数名称有自己的命名空间，所以$name、%name、@name 和函数 name 彼此之间没有什么关系，从而也不会造成混淆。

ParameterList 为函数参数表。Perl 5 中参数的使用与其他语言中有所不同，即不带参数表声明的函数可以在调用时不限制函数参数的个数和类型。对于带参数表定义的子程序则必须按照指定的参数调用，否则会出错，这一点与其他语言相似。声明的参数称为形参，调用子程序时所用的数值和变量称为实参。定义参数表时，就是在声明或定义参数时将每个参数的类型指示符放在括号中。每个类型指示符定义这个位置所需的参数。

紧跟在函数名称后面的语句块，就是函数的定义。当函数被调用时，会执行此块内的语句，若有返回值，执行结束后会传给调用者并返回。

请看下面的例子：

```
sub greeting{
    print "How do you do?\n";
}
```

也可以把函数定义放在程序代码的任何地方，Perl 的习惯是把它们放在主程序后面。

函数定义是全局（global）的，不能定义局部（local）函数。若有两个函数名称相同，后面的定义会把前面的覆盖掉，系统不会发出任何警告信息。

在函数定义中，可以存取任何位于程序其他部位的变量（它们被称为全局变量）。事实上，函数定义中参考到的变量，除非是私有变量（稍后会讨论），否则都会往“外面”去找。请看下面的例子：

```
sub greeting {
    print "Hello , $name.\n";
}
```

其中$name 会参考到全局的$name，它可能位于程序的其他地方。
也可以定义匿名函数如下：

```
$subReference = sub {code_block};
```

左边的变量$subReference 虽然可以省略，但如果要调用匿名函数，就必须有它。由于匿名函数是在赋值语句中生成的，闭大括号后面要有分号，以结束赋值语句操作。
下面是使用自定义函数的一个例子：

**程序清单 9-1：自定义函数的使用举例**

```
sub cut2fmt {
    my(@positions) = @_;
    my $template   = '';
    my $lastpos    = 1;
    foreach $place (@positions) {
        $template .= "A" . ($place - $lastpos) . " ";
        $lastpos   = $place;
    }
    $template .= "A*";
    return $template;
}
$fmt = cut2fmt(8, 14, 20, 26, 30);
print "$fmt\n";
```

程序输出结果为：

```
A7 A6 A6 A6 A4 A*
```

## 9.3 函数的调用

调用函数的方法有好多种，下面将分别介绍：

### 9.3.1 先定义后调用

```
sub subname {
    ...
}
...
subname();
```

这种调用自定义函数的方法很简单，只要在任何语句中，写出函数的名称，后面再加括号即可。请看下面的例子：

**程序清单 9-2：先定义后调用自定义函数**

```
sub sum_of_a_and_b {
    return $a + $b;
}
$a = 3;
$b = 4;
$c = sum_of_a_and_b();
print $c,"\n";
$d = 3*sum_of_a_and_b();
print $d;
```

程序输出为：

```
7
21
```

### 9.3.2 用&调用

在函数声明和定义之前调用它，可以采用这种方法：

```
&subname();
   ...
sub subname{
   ...
}
```

### 9.3.3 前向引用

前向引用方法先定义子程序名，后面再定义子程序体。这种方式和 C 语言很相似。

```
sub subname;
...
subname();
...
sub subname{
...
}
```

### 9.3.4 用 do 调用

采用下面的方式调用函数

```
do my_sub(1, 2, 3);
```

等同于下面的方式：

```
&my_sub(1, 2, 3);
```

但这种方式不是很常用。

### 9.3.5 函数调用的返回值

函数一定是语句的一部分，而其返回值则是 return 或最后一个语句的值。当然也可以没有返回值，例如前面介绍过的 greeting()函数就没有返回值，它相当于 Pascal 语言中的过程。

返回值可以是标量，也可以是列表。下面就分别举例说明这两种情形。

1. 返回值为标量

下面是一个例子：

```
sub sum {
    return $x + $y;
}
```

函数定义内最后一个语句的值是$x + $y 的值，所以该函数的返回值就是$x 和 $y 的和。下面是实际使用的情形：

```
$x = 6;
$y = 9;
$a = sum();      # 现在 $a = 15;
$b = 9 * sum() + $a;      # 现在 $b = 150;
```

2. 返回值为列表

函数也可以在列表环境下返回列表值。请看下面的例子：

```
sub list {
    return ($x, $y);
}
$x = 6;
$y = 9;
@a = list();        # 现在 @a = (6,9)
```

下面是一个返回值为列表的具体例子：

**程序清单 9-3：返回值为列表的自定义函数**

```
sub bigger_than {
    my($n,@values);        # create some local variables
    ($n,@values) = @_;        # split args into limit and values
    my(@result); # temporary for holding the return value
    foreach $_ (@values) {        # step through the arg list
        if ($_ > $n) {      # is it eligible?
            push(@result,$_);        # add it
        }
    }
       return @result;
 }
@list = (95..105);
@new = bigger_than(100,@list);
foreach $number(@new){
  print $number," ";
}
```

```
print "\n";
@this = bigger_than(5,1,5,122,-34.15,30,780);
foreach $number(@this){
   print $number," ";
}
```

程序输出为:

```
101 102 103 104 105
122 30 780
```

## 9.4 更方便地使用函数

读者可能要问，如果函数仅仅依赖全局变量，那岂不是很容易混淆，很容易弄错吗？因为调用时很可能忘记了全局标量的值，或者它的值已经改变了。除了在函数里使用全局变量的方法之外，Perl 5 当然也提供了其他的方法。下来将要介绍另外几种方法，说明函数无需依赖全局变量。

### 9.4.1 在函数中使用参数

在函数的声明中使用参数的方法我们前面已经作过简要的介绍，现在再讨论一下这种方法的使用。在 Perl 程序中，函数名称后面如果跟着用括号括起来的列表值，此列表在函数进行的过程中，会自动设定一个名为@_的变量，函数可以检查这个变量，得知传来的参数个数和值。请看下面的例子：

```
sub greeting {
    print "$_[0]!\n";     # 打印出第一个参数
}
```

在这个例子里用到$_[0]，即是@_的第一个元素。请注意，它和$_没有任何关系，不要混淆!

下面是一个具体调用的例子：

```
greeting ("How","are","you");
```

上面的程序会在屏幕上打印出：

"How!"

前面提到过，不带参数表声明的函数可以在调用时不限制函数参数的个数和类型，所以在这里读者看到 greeting 函数声明时并没有带参数，但这里调用时却出现了参数"Hello"，注意这一点可能与以前的经验有所不同。

传入的参数如果超过，Perl 会忽略这件事；如果试图存取没传入的参数（即超过了数组@_的范围），只会得到 undef，这和一般的数组一样。

变量@_对函数来说是私有的，若有个全局变量也叫@_，调用时系统会把它的值储存起来，等函数返回后再复原全局变量@_。

使用带参数的函数，就可以将上节的"x 加 y"函数改写如下：

**程序清单 9-4：使用带参数的函数**

```
sub sum {
    return $_[0] + $_[1];
}
print sum (6,9);        # 在屏幕上打印出15
$a = sum(3,4);          # 现在 $a = 7
```

从上面可以看出，使用参数使函数的调用非常方便了。

但是如果想和 C 语言一样，在函数中使用中间变量而又不想让它影响外面具有相同名字的全局变量时又该怎么办呢？下面我们就讨论这个问题。

## 9.4.2 在函数定义中使用私有变量

用户可以利用 my 操作符，自定义私有变量。使用的方法很简单，只要用合法的变量名当参数，该操作符就会产生以此为名的局部变量。

请看下面的例子：

**程序清单 9-5：在函数定义中使用私有变量**

```
sub   sum {
    my   ($sum);        # $sum声明为私有变量
    $sum = 0;
      foreach $_(@_) {
      $sum += $_;
   }
   return $sum;
}
@number = (1..100);
```

```
print sum(@number);
```

程序执行结果是在屏幕上输出：

```
5050
```

my ($sum)这条语句执行时，如果有$sum 这个全局变量的话，系统会把它暂时保存起来，并产生一个全新的变量，名为$sum，值为 undef。函数结束时，刚刚产生的局部变量随即消失，并“取出”全局变量$sum。有时候，函数会调用另一个函数，对被调用的函数来说，会看到调用者的局部变量，此时也可以用 my()操作符产生同名的局部变量。

请看下面的例子：

**程序清单 9-6：在函数定义中使用 my 操作符**

```
sub smaller_than_30
{
    my (@result);       # 局部变量，暂时储存返回值
    foreach $_ (@_)     # 扫描参数列
      {
       if ($_<30)       # 判断是否符合条件
       {
          push (@result, $_);      # 将符合条件的数加入列表中
          }
       }
       return @result;       # 返回最后的列表
}
@number = (20..40);
@num_list = smaller_than_30(@number);
foreach $num (@num_list){
       print $num," ";
  }
```

上面这个程序由传来的数组中，挑出所有小于 30 的元素。程序执行的结果为：

```
20  21  22  23  24  25  26  27  28  29
```

my 操作符的返回值是可设值的列表，所以可以放在数组设值操作符的左侧，其中每个元素的初始值都是 undef。事实上，此列表的元素就是“新”产生的局部变量，所以像这样的写法：

```
my ($n,@values);
($n,@values) = @_;
```

可以简写成：

```
my ($n,@values) = @_;
```

类似地，也可以这样替局部变量设初始值：

```
my ($sum) = 0;
```

还有一点值得注意的是，my 操作符虽然看起来只是用来声明变量，但确实是个可执行的操作符。

下面是另一个在函数中运用私有变量的例子：

**程序清单 9-7：在函数定义中使用私有变量**

```
sub func ($) {
    my $n = shift;
    print "you gave me $n\n";
}
func @foo;
func split /:/;
func "a", "b", "c";
func("a", "b", "c");
```

程序执行结果为：

```
you gave me 0
you gave me 0
you gave me a
```

### 9.4.3　使用半私有变量

如果在一个函数中调用了另外一个函数，而在被调用的函数中想要使用调用函数的私有变量，该怎么办呢？使用前面介绍的方法，被调用函数可以使用全局变量，也可以用参数调用，但是它都不能很好地满足要求。因此，Perl 提供了另一个产生私有变量的方法，那就是利用 local 函数。正确区分 my 和 local 之间的不同非常重要。

下面的例子能够说明这一点：

**程序清单 9-8：在函数定义中使用半私有变量**

```
sub greeting1{
   local ($hello) = "How do you do?";
   greeting2();
}
```

```
sub greeting2{
    print "$hello";
}
$hello = "How are you doing?";
greeting2();
greeting1();
greeting2();
```

该程序的执行结果如下：

```
"How are you doing?"
"How do you do?"
"How are you doing?"
```

如果在 greeting1 函数中用 my 代替 local，私有变量$hello 的作用范围仅限于该函数本身。读者从执行结果可以得知，用 local 产生的私有变量不再那么“私有”了，被 greeting1 调用的函数 greeting2 也能“看得见”它了。这就是说，用 local 产生的私有变量，其作用范围为所有在该变量产生处被调用的函数。

下面是使用 local 声明半私有变量的另外一个例子，从中大家可以更加清楚地看到 local 的妙用：

```
my @nums = (0 .. 5);
sub first {
    local $nums[3] = 3.14159;
    second();
}
sub second {
    print "@nums\n";
}
second();
first();
```

程序执行的结果为：

```
0 1 2 3 4 5
0 1 2 3.14159 4 5
```

Perl 中，一个函数可以调用另一个函数，被调用的函数还可以再调用其他的函数，这个过程可以一直持续，这就是函数的嵌套。唯一的限制是系统的内存空间。当函数调用它本身时，就成了递归调用。

下面就是一个嵌套调用函数的例子：

**程序清单 9-9：嵌套调用函数**

```
sub outer {
    my $x = $_[0] + 35;
    local *inner = sub { return $x * 19 };
    return $x + inner();
}
$number = 25;
print outer($number);
```

程序执行结果为：

```
1200
```

递归调用有以下两个条件：

（1）除了不被子程序改变的变量外，所有的变量必须是局部的。

（2）该子程序要含有停止调用本身的代码。



子程序中局部变量的操作符有两种：my 和 local．其区别是：my 定义的变量只在该子程序中存在；而 local 定义的变量不存在于主程序中，但存在于该子程序和该子程序调用的子程序中。

### 9.4.4 文件层级的 my 变量

my 操作符完全可以用在程序的最上层，不属于任何函数或代码块。这种用法虽然和上面的说法不同，不过非常有用，特别是配合 Perl pragma 时，如：

```
use strict;
```



pragma 是一种编译器命令，其他类似的命令包括定义整数运算、重载数学操作符，或产生更多出错信息……不胜枚举。读者若有兴趣，可参阅有关书籍。

如果在文件最前面加上这一行，除非先声明，否则不能使用任何变量。如果读者使用过 Visual Basic，可能会对 option explicit 有印象，这一行的作用与其类似。用 my 声明变量，像这样：

```
use strict;
my $x;              # 此时$x为undef
my @list = qw (apple banana orange);
... ...
```

```
@y = sort @list;
```

最后一行语句会引发编译错误，因为此语句用到了未先声明的变量@y，程序将无法执行。这种作法有两大好处：

- 程序会执行得比较快，因为存取用 my 声明的变量比一般变量快。
- 较易发现输入拼写上的错误，因为未声明的变量（打错的字）不能存取。

基于这些原因，建议读者在开始编写程序时，最好在最前面加上 use strict 语句。

## 9.5 Perl 5 中预定义的函数

Perl 5 中预定义了三个函数，分别在特定的时间执行，它们是：

- BEGIN 函数——在程序启动时被调用。
- END 函数——在程序结束时被调用。
- AUTOLOAD 函数——在找不到某个函数时被调用。

你可以自己定义它们，以在特定时间执行所需要的动作。如下面的例子所示：

```
  BEGIN {
      print("Hi! Welcome to Perl!\n");
  }
  AUTOLOAD{
      print("subroutine $AUTOLOAD not found\n");     # 变量$AUTOLOAD即未找到的子程序名
      print("arguments passed: @_\n");
  }
END{
      print ("The program has ended. Bye_bye!\n");
}
```

若同一个预定义函数被定义了多次，则BEGIN顺序执行，END逆序执行。

## 9.6 函数的应用举例

下面给出几个运用自定义函数解决问题的例子。

### 9.6.1 二进制和十进制之间的转换函数

下面的程序有两个函数，dec2bin 函数可将一个十进制的输入数，变成二进制数；bin2dec 函数将一个二进制的输入数，变成十进制数。这两个函数虽然行数不多，但却完成了其他语言需较多代码才能完成的功能，这充分说明了 Perl 在处理设计字符方面的强大。读者可以从下面的程序中仔细体会这一点。

**程序清单 9-10：二进制和十进制函数的转换**

```
sub dec2bin {
    my $str = unpack("B32", pack("N", shift));
    $str =~ s/^0+(?=\d)//;    # otherwise you'll get leading zeros
    return $str;
}
sub bin2dec {
    return unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}
$num = bin2dec('0110110');
$binstr = dec2bin(54);
```

程序执行结果为：

```
The decimal of '0110110' is : 54
The binary of '54' is : 110110
```

### 9.6.2 将角度转换为弧度

弧度和角度之间的相互转换在处理三角形时是经常要用到，下面的程序中有两个函数，deg2rad 函数可将角度转化为弧度；rad2reg 函数将弧度转化为角度。

**程序清单 9-11：将角度转化为弧度的函数**

```
$PI = 3.14159265358979;
sub deg2rad {
    my $degrees = shift;
    return ($degrees / 180) * $PI;
}
sub rad2deg {
    my $radians = shift;
```

```
    return ($radians / $PI) * 180;
}
print deg2rad(90),"\n";
print rad2deg(1);
```

程序执行结果为：

```
.5707963267949
57.2957795130824
```

### 9.6.3 求一个数的对数

下面的程序是求以任意数为底的数的对数问题，它以系统给定的 log 函数为基础，定义了任意底的对数的求法，十分方便。

**程序清单 9-12：对数函数**

```
sub log_base {
    my ($base, $value) = @_;
    return log($value)/log($base);
}
$answer1 = log_base(10, 10000);
print "log10(10,000) = $answer1\n";
$answer2 = log_base(3,81);
print "log10(3,81) = $answer2\n";
```

程序输出结果为：

```
log10(10,000) = 4
log10(3,81) = 4
```

### 9.6.4 数字和英文的对应

1. 将输入的数字变成对应的英文

下面定义了一个函数 card，以 1 至 9 的数字为参数，并返回其英文 (one、two)。若输入的数字超过范围或根本不是数字，则按原样返回。程序如下：

**程序清单 9-13：将数字变成英文**

```
sub card {
```

```
    my %card_map;
    @card_map{1..9} = qw( one two three four five six seven eight nine );
    my($num) = @_;
    if ($card_map{$num}) {
      $card_map{$num}; # return value
    } else {
      $num; # return value
    }
}
# driver the routine:
while (<>) {
    chomp;
    print "card of $_ is ", &card($_), "\n";
     }
```

程序执行情况如下：

```
9
card of 9 is nine
7
card of 7 is seven
0
card of 0 is 0
-3
card of -3 is -3
exit
card of exit is exit
```

2. 两数相加，输出英文结果

利用上面的函数，另外编写一个函数执行两数相加，并打印出类似“Two plus two equals four”这样的结果，特别说明第一个字母要大写。程序如下：

**程序清单 9-14：输出两个数相加的英文结果**

```
sub card { ...; } # from previous problem
print "Enter first number: ";
chomp($first = <STDIN>);
print "Enter second number: ";
chomp($second = <STDIN>);
```

```
$message = &card($first) . " plus " . &card($second) . " equals " . &card($first+$second) . ".\n";
print "\u$message";
```

该程序的第一句只是一个示意，表示借用上面的函数 card。实际执行该程序时，应该将上面的函数完整的写出来，否则会出错。该程序执行情况如下：

```
Enter first number: 9
Enter second number: 6
Nine plus six equals 15.
```

3. 可接受两负数相加，输出英文

将上面两个程序改变使其接受的范围扩充为-9 到+9，下面是改写后的 card 函数：

**程序清单 9-15：接受两个负数相加并输出英文结果**

```
sub card {
  my %card_map;
  @card_map{0..9} = qw( zero one two three four five six seven eight nine );
  my($num) = @_;
  my($negative);
  if ($num < 0) {
    $negative = "negative ";
    $num = - $num;
  }
  if ($card_map{$num}) {
    $negative . $card_map{$num};      # return value
  } else {
    $negative . $num;      # return value
  }
}
```

加上程序清单 9-14 中的测试语句后，程序执行情况如下：

```
Enter first number: -9
Enter second number: -6
Negative nine plus negative six equals negative 15.
```

### 9.6.5 一个较复杂的例子

下面的程序中包含一个函数 commify_series，它将一个特殊的列表（可以看作是多维数

组）按照逗号分开，然后将用逗号分开的各项又用 and 连接起来。该函数里面用到了条件表达式和两个系统函数：grep 函数和 join 函数，充分体现了 Perl 在处理字符问题上的方便。程序给出了一个列表，并写了一段代码对函数进行了测试。

**程序清单 9-16：一个使用较复杂函数的例子**

```
@lists = (
    [ 'Just one thing' ],
    [ qw(Mutt Jeff) ],
    [ qw(Peter Paul Mary) ],
    [ 'To our parents', 'Mother Theresa', 'God' ],
    [ 'pastrami', 'ham and cheese', 'peanut butter and jelly', 'tuna' ],
    [ 'recycle tired, old phrases', 'ponder big, happy thoughts' ],
    [ 'recycle tired, old phrases', 'ponder big, happy thoughts', 'sleep and dream peacefully' ],
    );
sub commify_series {
    my $sepchar = grep(/,/ => @_) ? ";" : ",";
    (@_ == 0) ? ''                                          :
    (@_ == 1) ? $_[0]                                       :
    (@_ == 2) ? join(" and ", @_)                           :
                join("$sepchar ", @_[0 .. ($#_-1)], "and $_[-1]");
}
foreach $aref (@lists) {
    print "The list is: " . commify_series(@$aref) . ".\n";
}
```

程序的执行输出为：

```
The list is: Just one thing.
The list is: Mutt and Jeff.
The list is: Peter, Paul, and Mary.
The list is: To our parents, Mother Theresa, and God.
The list is: pastrami, ham and cheese, peanut butter and jelly, and tuna.
The list is: recycle tired, old phrases and ponder big, happy thoughts.
The list is: recycle tired, old phrases; ponder big, happy thoughts; and sleep and dream peacefully.
```

## 9.7 小结与练习

### 9.7.1 小结

本章介绍的函数又称为子程序。函数对于处理一类相似的问题，避免重复性工作具有重大意义。当大量的任务相似时，一般需要将对它们的处理写成函数，每次只需输入必要的参数调用函数即可。函数还具有封装的功能，它将执行一段特定功能的代码封装起来，只提供给外部接口，而不需要了解它的内部状况，当需要改变时，只需对函数内部进行修改即可，无需在外部进行改动（这样往往会造成程序的大规模变动，增大工作量），所以函数还有利于程序的维护。此外，封装的函数使程序的模块化程度提高、结构紧凑，逻辑显得十分清晰，可读性强。

本章第一节给出了一些常用的系统函数的列表。了解这些函数的功能和用法对我们是很有好处的。

第二节介绍了如何定义用户自己的函数。

第三节介绍了调用函数的几种方法。最常用的方法是先定义，后调用。本节还介绍了函数的返回值，函数的返回值可以为标量，也可以为列表，读者尤其要注意返回值为列表的情形，因为这实际上相当于返回了多个标量值。

第四节介绍了使用函数时需要掌握的知识，包括在函数中使用参数，在函数中使用私有变量和半私有变量等。

第五节简要介绍了系统预定义的几个函数的几种方法。读者只需对此作一般性的了解即可。

本章最后一节用几个例子介绍了函数的应用。这些例子都是读者经常要碰到的函数问题。包括十进制数和二进制数的相互转换，角度和弧度的相互转换，求一个数的对数等。最后还给出了一个较为复杂的例子。通过这些例子，读者应该对函数的用处有一个更直观的认识了。

学习了本章，读者应该了解一些常用的系统函数，掌握怎样定义和调用自己的函数。下一章将介绍引用。

### 9.7.2 习题与思考

（1）写一个程序，输出 0 到 45 度角的余弦和反余弦值。提示：系统提供了余弦函数，但没有提供反余弦函数，所以反余弦函数需要自己定义。

（2）写一个函数，求一列数的最大值，并写一段代码对该函数进行测试。

（3）写一个函数，使它能够接受一个列表，并将列表中的所有小写字母变成大写字母。

（4）写一个程序，输出当天的日期（年月日）和当天是当年的第几天。

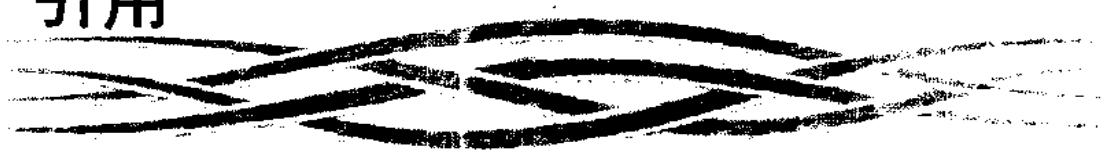
（5）试写出下面程序的输出结果。

```
$bree = 361535725;        # 16 Jun 1981, 4:35:25
$nat  =  96201950;        # 18 Jan 1973, 3:45:50
$difference = $bree - $nat;
print "There were $difference seconds between Nat and Bree\n";
$seconds  =  $difference % 60;
$difference  =  ($difference - $seconds) / 60;
$minutes  =  $difference % 60;
$difference = ($difference - $minutes) / 60;
$hours  =  $difference % 24;
$difference = ($difference - $hours) / 24;
$days  =  $difference % 7;
$weeks  = ($difference - $days) / 7;
print "($weeks weeks, $days days, $hours:$minutes:$seconds)\n";
```

（6）写一个程序，求直角三角形的斜边。

# 第10章

## 引用



引用简介
引用变量类型简介
直接引用变量的使用
访问引用的其他方法
小结与练习



引用数据类型是 Perl 5 的一个新功能。在引用出现之前，很难在 Perl 中生成多维数组和包含其他数据结构的数据结构。引用能够支持复杂的数据对象，这比引用的效率更加重要。在 Perl 中加进了复杂数据对象，使 Perl 从一流的字处理语言变成了能解决各种问题的世界级语言。

生成引用并不难，但使用引用则比较复杂。

本章要点：

- 直接引用变量的使用。
- 匿名引用变量的使用。
- 几种访问引用的操作符。
- 用引用构建多维数组。

# 10.1 引用简介

## 10.1.1 什么是引用

引用就是指针，它可以跟踪变量在内存中的存放位置，它可以指向变量、数组、散列甚至函数。了解 C 语言或 Pascal 语言的程序员应该对引用（即指针）的概念不太陌生，引用就是某值的地址，对它的使用则取决于程序员和语言的规定。在 Perl 中，也可以把引用称为指针，二者是通用的。

生成普通变量时，实际上是生成一个数据的存储位置。这就像在厨房中存放各种食物的容器位置，声明一个标量变量、散列或普通数组可以看作是选择合适类型的存储容器。普通变量与引用变量的差别在于：引用变量中存放的东西不是实在的内容而是存放这些内容的地址。

总之，引用跟踪内存中的变量存放地址，它可以跟踪任何类型的变量。引用变量总是一个标量变量，包含另一个对象的地址。无论是存放一个标量变量的地址，还是存放大型对象清单的地址，引用变量都只占用计算机中的一个地址空间。

## 10.1.2 引用操作符

可以使用引用操作符 \、[]、{}、sub {;} 和 bless 生成引用变量（引用操作符 bless 是个特例，不是严格意义上的引用操作符）。

表 10-1 列出了生成引用的不同方法。

表 10-1　　Perl 5 中的几种引用操作符

| 数据类型 | 操作符 | 例子 |
| --- | --- | --- |
| Scalar | \$var | $car = \$Porsche; |

续表

| 数据类型 | 操作符 | 例子 |
| --- | --- | --- |
| Array | \@array | $allStores[$number]= \@store_Invertory; |
| Hash | \%hash | $storeType = \%stores; |
| File handle | *FILEHANDLE | $inputReference = *STDIN; |
| Constant | \literal | $pi = \3.14159; |
| Subroutine | \&subRoutine | $callBack = \&numericSort; |
| Symbolic | \$variableName | $var = "var2"; $var2Ref = \$var; |
| Anonymous array | [LIST] | $colors = ["GREEN","RED","BLUE"]; |
| Anonymous hash | {List of name / value pairs} | $hexCodes = {GREEN->0X00FF00, RED->0XFF0000, BLUE->0X0000FF } |
| Anonymous subroutine | sub { }; | $greeting = sub {print "$hello";}; |
| Object reference | bless $self; | sub new {self = { }; bless $self; return $self; } |

下面是使用引用操作符的一些例子：

```
$scalarref = \$foo;
$constref  = \186_282.42;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
$globref   = \*STDOUT;
```

## 10.2　引用变量类型简介

引用变量有六种类型，它由引用变量中存放的地址类型确定：

- Direct reference——直接引用。

- Constant——常量引用。
- Subroutine——函数引用（函数引用）。
- Symbolic——符号引号。
- Anonymous——匿名引用。
- Object——对象引用。

### 10.2.1 直接引用变量

顾名思义，直接引用变量就是直接引用其他变量，包括标量、数组、散列、文件句柄和其他引用变量。直接引用变量可以修改引用的变量内容。直接引用变量用反斜杠操作符（\）表示。

直接引用变量是最常用的引用方法，后面我们将对它的使用方法作详细的介绍。

### 10.2.2 常量引用变量

常量引用变量也使用反斜杠操作符，但指的是直接数而不是变量。可以修改常量引用变量，但不能修改常量引用变量所指的直接数。直接数通常保存在特殊的保护内存中。例如，计算机内存中只保存一份直接数 10，则不允许修改 10 的值。

```
$pi = \3.14159;
$myname = \"Charlie";
```

### 10.2.3 函数引用变量

函数引用变量是直接引用变量与常量引用常量的混合，不能用函数引用变量修改函数引用变量所指的函数，但可以修改函数本身。生成函数引用变量时，必须用&操作符告诉 Perl 5 正在引用函数而不是直接数。

Perl 中函数的引用与 C 语言中函数的指针类似，构造方法如下：

```
$pointer_to_sub = sub {... declaration of sub ...};
```

通过所构造的引用调用函数的方法为：

```
&$pointer_to_sub(parameters);
```

函数的返回值不仅限于数据，还可以返回函数的引用。返回的函数在调用处执行，但却是在最初被创建的调用处被设置。

下面就是一个使用函数引用变量的例子：

**程序清单 10-1：函数引用变量的使用**

```
sub counter_maker {
    my $start = 0;
    return sub {
        return $start++;
    };
}
$counter1 = counter_maker();
$counter2 = counter_maker();
for ($i = 0; $i < 5; $i ++) {
    print &$counter1, "\n";
}
print &$counter1, " ", &$counter2, "\n";
```

程序输出结果如下：

```
0
1
2
3
4
5 0
```

读者可以仔细分析结果，体会函数引用的作用。

### 10.2.4　符号引用变量

符号引用变量是 Perl 5 中独有的。符号引用变量是包含另一变量的直接数名称的变量。我们用反斜杠操作符和变量名指定引用符号。这个机制比较复杂，很容易使代码出错。

下面的例子用符号引用变量生成变量而不用事先声明或定义：

**程序清单 10-2：符号引用变量的使用**

```
$color = "red";
$$color = 0xFF0000;
print "$red\n";
$color = "blue";
$$color = 0x0000FF;
print "$blue\n";
```

程序执行结果是在屏幕上打印出：

```
0xFF0000
0x0000FF
```

第一条语句将直接数“red”赋给了标量变量$color，第二条语句则把标量变量$color 当作引用变量，将十六进制值 0xFF0000 放进颜色所指的变量。可以看出，标量变量$color 并不包含变量引用，而是包含直接数“red”。标量变量$color 所指的字符串在用作变量时即被定义成了变量。

在 Perl 5 中，变量随着被使用就生成了。这里$$color 把$color 中包含的值作为变量，这从第三行语句可以看出来，该行要求打印变量$red 的值，即十六进制值 0xFF0000，表示计算机上的红色。下面的三行与上面相似，打印出绿色的十六进制值。

在 Perl 5 中，符号引用发生在访问包含字符串的标量变量时。字符串可能是现有变量名或在引用时成为新变量。

### 10.2.5 匿名引用变量

匿名引用变量允许生成不命名的对象引用，这些对象也称为孤立对象。匿名引用操作符（[ ]、{}和 sub{;}）是上下文相关的，只在对标量变量赋值时生成引用。匿名函数操作符（sub {;}）要求在函数定义内用分号结尾。Perl 5 把匿名引用变量当作在内存中有特定地址的语句。

所谓匿名数组与散列就是没有数组名的数组和散列，下面就简要介绍一下它们的使用。

通常，生成数组时是对指定数组的单元赋值，例如：

```
@arrayName = (1,2,3);
```

可以生成匿名数组如下：

```
$arrayReference = [1,2,3];
```

然后数组引用可以存进另一数组中，生成多维数组，如下所示：

```
$arrays[0] = $arrayReference ;
```

通常生成多维数组时，将一个数组的引用赋值给另一数组的单元，如下所示：

```
@codes[1] = \@codeNumbers;
```

这里首先要生成数组@codeNumbers，然后再被赋值给@codes 数组。这里左边之所以要使用@符号，而不是$符号，是因为元素 codes[1]本身也是一个数组。

而使用匿名数组可以通过使用匿名数组操作符（[ ]），跳过生成中间数组的步骤，如下所

示：

```
@codes[1] = [1,2,3,4,5,6,7,8,9,10];
@codes[2] = [a,b,c,d,e,f,g,h,i,j,k];
```

使用匿名数组引用操作符和赋值操作符时，就是要对赋值的标量变量生成匿名数组引用。任何使用数组引用操作符（\@arrayName）的场合都可以利用匿名数组引用操作符。初始化时可以将匿名数组引用操作符作为操作符对待。

匿名散列操作符（{}）使用的规则与匿名数组引用操作符（[ ]）完全相同。可以在通常出现散列操作符（\%hash）的场合使用匿名散列操作符。

下面就是使用匿名散列的几个例子：

```
$hashref = {    # 普通的散列引用
    'Adam'    => 'Eve',
    'Clyde'   => $bonnie,
    'Antony' => 'Cleo' . 'patra',
};
$table1 = {    # 散列中嵌套着数组的散列引用
              "john" => [ 47, "brown", 186 ],
              "mary" => [ 23, "hazel", 128 ],
              "bill" => [ 35, "blue",   157 ],
};
$table2 = {     # 散列中嵌套着散列的散列引用
            "john" => { age      => 47,
                        eyes     => "brown",
                        weight => 186,
                      },
            "mary" => { age      => 23,
                        eyes     => "hazel",
                        weight   => 128,
                      },
            "bill" => { age      => 35,
                      eyes     => "blue",
                        weight => 157,
                      },
};
```

下面是一个使用匿名函数引用的例子：

**程序清单 10-3：匿名函数引用的使用**

```
$coderef = sub { print "Boink!\n" };
print &$coderef ;
```

程序执行结果为：

```
Boink!
```

### 10.2.6 对象引用变量

对象引用变量也就是匿名散列引用。Perl 5 提供有 bless 操作符，用来生成引用对象与生成对象的类之间的链接（类是包含相关函数的包）。对象引用变量通常在函数 new 中生成：

```
sub new
{
    $self = {};      # 产生一个匿名散列
    bless $self;     # 将散列与类相联系
    return $self;    # 返回对散列的引用
}
```

## 10.3 直接引用变量的使用

有多种构建引用的方法，几乎可以对任何数据建立引用，如简单变量、数组、函数、文件句柄等。

如果$pointer 的值为一个数组的指针，则通过形式@$pointer 来访问数组中的元素。形式@$pointer 的意义为“取出$pointer 中的地址值当作数组使用”。类似的，%$pointer 为指向散列表中第一个元素的引用。

下面，将分别举例说明各种常用的使用引用的方法。

### 10.3.1 使用标量引用

一般是用反斜线（\）来创建变量的引用，反斜线操作符与 C 语言中传递地址的操作符&的功能类似。下面是创建标量变量引用的例子：

```
$variavle = 47;
$pointer = \$variable;
```

```
$ice = "jello";
$iceprt = \$ice;
```

引用$pointer 指向存有$variable 值的位置；引用$iceprt 指向"jello"。即使最初的引用$variable 被销毁了，仍然可以通过$pointer 访问该值。

在上面的例子中，引用变量$pointer 保存的是$variable 的地址，而不是值本身，要获得值，形式为两个$符号，如下例所示：

**程序清单 10-4：使用标量引用**

```
$value = 36;
$pointer = \$value;
printf "\n Pointer Address $pointer of $value \n";
printf "\n What Pointer *($pointer) points to $$pointer\n";
```

结果输出如下：

```
Pointer Address SCALAR(0x806c520) of 36
What Pointer *(SCALAR(0x806c520)) points to 36
```

每次运行，输出结果中的地址会有所改变，但仍然可以看到$pointer 给出地址，而$$pointer 给出$variable 的值。

看一下地址的显示，SCALAR 后面一串十六进制，说明该地址指向标量，后面的数字是实际存储值的地址。

从上面的例子中我们可以看出，可以用反斜线“\”来生成引用，而用$号来取得标量引用里的数值（对于数组用“@”号，散列用“%”号等，后面将会讨论）。



指针就是地址，通过指针可以访问该地址处存储的数据。如果指针指向了无效的地址，就会得到不正确的数据。通常情况下，Perl 会返回 NULL 值提示出错信息，但不该依赖这一点，一定要在程序中把所有的指针正确地初始化，指向有效的数据项。

简单地说，每个$号都是一个数值容器。Perl 5 从右到左或由内向外机械地解释$号。每次遇到$号时，它把$号右边的对象当作一个数值容器。例如上例中的$$pointer 在解释器中就是分两步读取：

第一步：从最右边或最里边的$号开始，表示$号后面的对象是一个数值容器。Perl 5 取得对象$pointer 中的值，即一个地址；

第二步：第二个对象也有一个$号，表示这个对象也是一个数值容器。由于第一个对象包含第二个对象（地址），后者又是数值容器，因此，Perl 5 取得所取的第一个对象（地址）中包括（即指向）的数值。

实际上，上述过程可以无限深入。

### 10.3.2 使用数组引用

数组引用的建立与标量引用类似，只是数组引用在建立和使用时采用的不是“$”符号，而是“@”符号。

建立数组引用的语法如下：

```
$reference = \@arrayname;
```

访问数组引用的语法如下：

```
@$reference;
```

下面的例子具体地说明了直接数组引用的用法：

**程序清单 10-5：直接数组引用的使用**

```
@fruit = {"apple","banana","peach","orange"};
$fruitRef = \@fruit;
print "The fruits are @$fruitRef.";
```

执行的结果是在屏幕上打印出：

```
The fruits are apple banana peach orange.
```

访问数组的各个单元与访问整个数组一样简单。数组单元是标量变量，只要记住这个规则，就能知道如何访问数组单元了。通常访问方法如下：

```
$array[i];      #i代表要访问的元素在数组中的索引，即位置。
```

要在变量是直接数组引用时访问数组单元，首先要访问直接数组引用变量，然后把它当作普通数组处理，如：

```
$$arrayRef[0];
```

使用数组引用时，下面的几种写法是等价的：

```
$$arrayref  [2] = "Dorian";     #1
```

```
${ $arrayref }[2] = "Dorian";      #2
$arrayref->[2] = "Dorian";      #3
```

还可以以下面的方式使用数组引用，这些都是一些技巧：

```
@reflist = (\$s, \@a, \%h, \&f);        # List of four references
@reflist = \($s,   @a    %h,   &f);        # Same thing
@reflist = \(@x);      # Interpolate array, then get refs
@reflist = map { \$_ } @x;      # Same thing
@reflist = \(@x, (@y));      # But only single aggregates expand
@reflist = (\@x, map { \$_ } @y);      # Same thing
@envrefs = \@ENV{'HOME', 'TERM'};      # Backslashing a slice
@envrefs = \( $ENV{HOME},   $ENV{TERM} );      # Backslashing a list
@envrefs = ( \$ENV{HOME}, \$ENV{TERM} );      # A list of two references
```

### 10.3.3　使用散列引用

数组引用的规则也适用于散列引用。
生成散列引用的语法如下：

```
$reference = \%hashname;
```

访问散列引用的方式如下：

```
%$reference;
```

访问散列引用单元的方式如下：

```
$$reference{'RED'};
```

散列引用单元的赋值如下：

```
$$reference{'RED'} = 0xff0000;
```

对散列使用反斜线操作符的方法是一样的——即把所有散列名替换成引用$poniter。
下面几种使用散列引用的方式是等价的：

```
$$hashref  {KEY} = "F#major";      #1
${ $hashref }{KEY} = "F#major";     #2
$hashref->{KEY} = "F#major";      #3
```

下面的例子可以很好地说明这些问题：

**程序清单 10-6：散列引用的使用**

```
%weekday = ('1' => 'Mon', '2' => 'Tue', '3' => 'Wed',
                    '4' => 'Thu', '5' => 'Fri', '6' => 'Sat', '7' => 'Sun',
                   );
$pointer = \%weekday;
for($j = 0;$j<8;$j++) {
      $i = $j;
      print   '$$pointer{$i} is ';
      print   "$$pointer{$i} \n";
      print   '${$pointer}{\$i} is ';
      print   "${$pointer}{$i} \n";
      print   '$pointer->{\$i} is ';
      print   "$pointer->{$i}\n";
      # These next two lines should not show anything
      print   '${$pointer{\$i}} is ';
      print   "${$pointer{$i}}          ";
      print   '${$pointer->{\$i}} is ';
      print   "${$pointer->{$i}}\n";
}
```

结果输出如下：

```
$$pointer{$i} is Mon
${$pointer}{$i} is Mon
$pointer->{$i} is Mon
${$pointer{$i}} is          ${$pointer->{$i}} is
$$pointer{$i} is Tue
${$pointer}{$i} is Tue
$pointer->{$i} is Tue
${$pointer{$i}} is          ${$pointer->{$i}} is
$$pointer{$i} is Wed
${$pointer}{$i} is Wed
```

```
$pointer->{$i} is Wed
${$pointer{$i}} is        ${$pointer->{$i}} is
$$pointer{$i} is Thu
${$pointer}{$i} is Thu
$pointer->{$i} is Thu
${$pointer{$i}} is        ${$pointer->{$i}} is
$$pointer{$i} is Fri
${$pointer}{$i} is Fri
$pointer->{$i} is Fri
${$pointer{$i}} is        ${$pointer->{$i}} is
$$pointer{$i} is Sat
${$pointer}{$i} is Sat
$pointer->{$i} is Sat
${$pointer{$i}} is        ${$pointer->{$i}} is
$$pointer{$i} is Sun
${$pointer}{$i} is Sun
$pointer->{$i} is Sun
${$pointer{$i}} is        ${$pointer->{$i}} is
```

可以看到，前三种形式的输出显示了预期的结果，而后两种则没有。如果读者不清楚是否正确时，就输出结果看看。在 Perl 编程中，遇到有不明确的代码就用 print 语句输出来验证一下，这能使读者更清楚的了解 Perl 是怎样解释代码的。

从上面的例子可以看出，与数组类似，通过引用访问散列表的元素形式为 $$pointer{$index}，当然，$index 是散列表的键值，而不仅是数字。



关于 Perl 语言应该记住的最重要的一点是：Perl 中的数组和散列表始终是一维的。因此，数组和散列表只保存标量值，不直接存储数组或其他的复杂数据结构。

### 10.3.4　文件句柄的引用

有时，必须将同一信息输出到不同的文件中。例如，某程序可能在一个实例中输出到屏幕，另一个输出到打印机，再一个则输出到记录文件，甚至同时输出到这三个文件。相比较于每种处理写一个单独的语句，应该还有更好的实现方式，比如：

```
spitOut(\*STDIN);
spitOut(\*LPHANDLE);
spitOut(\*LOGHANDLE);
```

其中函数 spitOut 的代码如下：

```
sub spitOut {
   my $fh = shift;
   print $fh "Gee Wilbur, I like this lettuce\n";
}
```

其中文件句柄引用的语法为*FILEHANDLE。

## 10.4 访问引用的其他方法

引用可能很复杂，但生成引用和访问引用的规则是相同的，而不论其多么复杂或引用哪种数据类型。引用变量总是一个标量，可以用$号访问，然后再用要引用的变量的类型指示符（$、@、%）访问引用变量包含的变量值。

如果引用变量包含标量变量，则用$号取得标量变量的值；如果引用变量包含数组，则用@号或相应语法引用数组单元和其他数组变量；如果引用变量包含散列，则用散列语法等等。

$是通用的访问操作符，但有时这些$号不太好看，$$arrayRef[0]这样的表达式有时简直让人眼花缭乱。下面将要介绍几种引用操作符可以作为$号的一种替换，但有时它们是左结合的，而不像$号一样是右结合的。这些访问操作符包括大括号（{}）、相邻中括号（[ ][ ]）和向右箭头操作符（->）。

### 10.4.1 使用大括号

大括号（{}）能使代码更整洁。大括号用于变量名时，对代码没有实际影响，只是便于你和其他编程人员阅读代码。

如果大括号放在引用变量周围，则引用变量返回所包含的对象，执行大括号外的变量操作符。数组索引变量$#$arrayRef 变成$#{$arrayRef}，而数组变量@$arrayRef 变成@{$arrayRef}。大括号表示其中的变量是在大括号之外的操作符之前执行的项目。

如果记不住数组引用指向数组单元还是数组名，可以使用大括号，如数组单元引用$$arrayRef[2]变成${$arrayRef}[2]。首先进行大括号内的数组引用，然后读取数组单元 2 的值。

也可以用大括号表示多维数组的访问。但访问多维数组的常用的方法是向右箭头操作符。

### 10.4.2　使用向右箭头操作符

向右箭头操作符是常用于多位数组的引用操作符。

向右箭头操作符与访问$号相似，但与$号不同的是，向右箭头操作符是左结合的（左结合表示操作符影响其左边的符号）。规则如下：向右箭头操作符左边的项目是要访问的引用变量，也就是说向右箭头操作符访问其左边的对象，即$$arrayRef[2]变成$arrayRef->[2]。

向右箭头操作符允许$arrayRef->[2]语法。数组单元 0（$array[0]）中的值被访问并当作数组处理，然后再访问被访问数组中单元 2 的值。

### 10.4.3　使用方括号

另一种访问数组的方法是使用方括号，如：$array[0][2]。由于 Perl 5 知道数组单元中只能存放数组引用，不能存放数组，因此它假设第一个数组单元应为一个对数组的引用。即方括号之间和多维数组引用之间不需要用向右箭头操作符。

即以下三种形式是等价的：

```
${$array[0]}[2]
$array[0]->[2]
$array[0][2]
```

### 10.4.4　多维数组

作为上面介绍的操作符的一个例子，我们现在介绍一下多维数组。Perl 5 中的多维数组实际上是包含另一个数组引用的数组。下面就以三维数组为例来介绍一下多维数组。

下面的语句是用于画图的三维数组的例子：

```
$line = ['solid' , 'black' , ['1','2','3'] , ['4','5','6']];
```

此语句建立了一个含四个元素的三维数组，变量$line 指向该数组。前两个元素是标量，存储线条的类型和颜色，后两个元素是匿名数组的引用，存储线条的起点和终点。访问其元素语法如下：

```
$arrayReference->[$index]     # 访问一维数组
$arrayReference->[$index1][$index2]     # 访问二维数组
$arrayReference->[$index1][$index2][$index3]     # 访问三维数组
```

下面是一个实际的例子：

**程序清单 10-7：用引用产生多维数组**

```
$line = ['solid', 'black', ['1','2','3', ['4', '5', '6']]];
print "\$line->[0] = $line->[0] \n";
print "\$line->[1] = $line->[1] \n";
print "\$line->[2][0] = $line->[2][0] \n";
print "\$line->[2][1] = $line->[2][1] \n";
print "\$line->[2][2] = $line->[2][2] \n";
print "\$line->[2][3][0] = $line->[2][3][0] \n";
print "\$line->[2][3][1] = $line->[2][3][1] \n";
print "\$line->[2][3][2] = $line->[2][3][2] \n";
print "\n";
```

执行结果为在屏幕上输出如下信息：

```
$line->[0] = solid
$line->[1] = black
$line->[2][0] = 1
$line->[2][1] = 2
$line->[2][2] = 3
$line->[2][3][0] = 4
$line->[2][3][1] = 5
$line->[2][3][2] = 6
```

访问第三层元素的方式形如$line->[2][3][0]，类似于 C 语言中的 Array_pointer[2][3][0]。本例中，下标均为数字，当然亦可用变量代替。



多维数组容易混淆，所以建议你在使用多于三维的数组前，最好考虑使用其他数据结构来简化代码。

### 10.4.5 利用引用传给函数多个数组以及返回多个数组或散列

大家知道，传参数给函数时，我们最多只能传给它一个数组，因为函数在使用参数时只会用到@_，但是使用引用却可以克服这些限制，用法如下例所示：

**程序清单 10-8：利用引用传给函数多个数组**

```
@firstArray = (1..5);
@secondArray = ("A".."E");
firstSub( \@firstArray, \@secondArray);
```

```
sub firstSub {
    my($ref_firstArray, $ref_secondArray) = @_ ;
    print("The first array is   @{$ref_firstArray}.\n");
    print("The second array is @{$ref_secondArray}.\n");
}
```

程序执行的结果为：

```
The first array is   1 2 3 4 5.
The second array is A B C D E.
```

利用引用还可以让一个函数返回多个数组和散列，这时函数的功能大大扩展了，可以说引用使函数的返回值范围扩大了许多。下面程序中的函数 somefunc 正是利用了以上的原理，返回了一个数组和一个散列。

**程序清单 10-9：利用引用返回一个数组和一个散列**

```
($array_ref, $hash_ref) = somefunc();
sub somefunc {
    my @array;
    my %hash;
    @array = (0..5);
    %hash = (1=>"Jan",2=>"Feb",3=>"Mar");
    return ( \@array, \%hash );
}
print "The element of the array is :\n@$array_ref\n";
@hash_Array = %$hash_ref;
print "The element of the hash is : \n";
$i = 0;
foreach $ele (@hash_Array) {
   $i++;
   print $ele;
   if ($i % 2 == 1) {
        print " => ";
       next;
     }
     print "\n"
 }
```

程序的输出为：

```
The element of the array is :
0 1 2 3 4 5
The element of the hash is :
1 => Jan
2 => Feb
3 => Mar
```

## 10.5 小结与练习

### 10.5.1 小结

本章介绍了引用，它是 Perl 5 的一个重要的新功能。引用通常可以提高编程效率，因为在子程序之间传递数组地址比传递数组本身快得多。当然，由于现在的处理器速度极快，只有最复杂的问题才会要求引用的效率。数组引用和所有复杂结构一样，只有需要时才用，当遇到复杂问题时，如编码电子数据格式以转换遗留数据格式时再使用。

本章第 1 节首先介绍了引用的概念。引用是一个标量变量，包含有另一变量的地址。Perl 5 引用可以指向数据结构。然后重点介绍了各类引用操作符。

第 2 节简单地介绍了引用变量的六种类型—直接、常量、函数、符号、匿名和对象引用变量。

第 3 节举例详细介绍了直接引用变量的使用。访问操作符“$”，应放在要访问的变量的类型指示符的前面。用$号作为访问操作符是因为引用变量。$号访问标量变量，类型指示符（$、@、%和&）访问引用变量所指的变量。

第 4 节介绍了 Perl 5 提供的其他操作符，访问数组和散列等的引用。熟悉 C 语言的编程人员可能喜欢用方括号模拟多维数组，如$array[0][0]，但面向对象编程人员可以用向右箭头操作符，如$array[0]->[0]。向右箭头操作符是左结合的，因此首先访问左边的数组单元。

学习了本章，读者应该掌握 Perl 中引用的用法。下一章将介绍文件系统。

### 10.5.2 习题与思考

（1）写出下面程序的输出结果。

```
@AoA = (
        [ "fred", "barney" ],
```

```
            [ "george", "jane", "elroy" ],
             [ "homer", "marge", "bart" ],
              );
for($i = 0;$i < 3;$i++) {
    for($j = 0;$j < 3;$j++) {
      print $AoA[$i][$j],"   ";
    }
print "\n";
}
```

（2）写出下面程序的执行结果。

```
%recordOne = ( "Name"      => "Jane Hathaway",
               "Address" => "123 Anylane Rd.",
               "Town"      => "AnyTown",
               "State"     => "AnyState",
               "Zip"       => "12345-1234"
);
%recordTwo = ( "Name"      => "Kevin Hughes",
               "Address" => "123 Allways Dr.",
               "Town"      => "AnyTown",
               "State"     => "AnyState",
               "Zip"       => "12345-1234"
);
@database = ( \%recordOne, \%recordTwo );
print( %{$database[0]}->{"Address"} . "\n");
print( %{$database[1]}->{"Address"} . "\n");
```

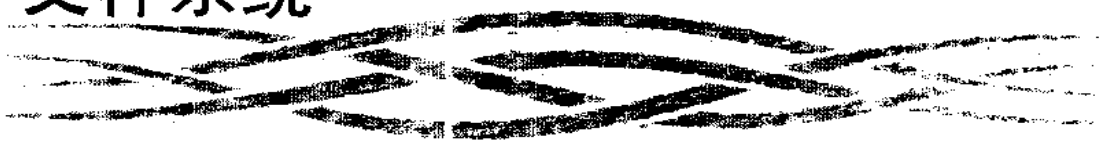
（3）写出下面程序的执行结果。

```
$scalar = 10;
@array  = (1, 2);
%hash   = ( "1" => "Davy Jones" );
printRef( $scalar, @array, %hash );
printRef( \$scalar, \@array, \%hash );
printRef( \&printRef, \\$scalar );
sub printRef {
    foreach (@_) {
```

```
        $refType = ref($_);
        defined($refType) ? print "$refType " : print("Non-eferenCe ");
    }
    print("\n");
}
```

# 第11章

# 文件系统



文件操作
目录操作
更改文件和目录的权限、所有权和时间戳
使用 DBM 文件
小结与练习



本章将介绍如何处理文件和目录，注意这里是指文件和目录本身，而非其中的数据。本章所讲的函数多数使用了 UNIX 操作系统的特性，在非 UNIX 系统中，一些函数可能没有定义或有不同的工作方式，使用时请查看 Perl 联机文档。

本章要点：

- 对文件操作的函数的使用。
- 对目录操作的函数的使用。
- DBM 文件的使用。

# 11.1 文件操作

## 11.1.1 打开/关闭文件

打开文件的语法为：

```
open (filevar, filename);
```

其中 filevar 为文件句柄，也就是程序中用来代表某文件的代号，它与文件名相关联。文件句柄遵循变量名的命名规则，但不是以$开头。文件句柄全部使用大写字母，以区别于 Perl 5 中的全部用小写保留字。filename 为文件名，其路径可以是相对路径，也可以是绝对路径如：

```
open(FILE1,"file1");     # 使用相对路径
open(FILE1, "/u/jqpublic/file1");     # 使用绝对路径
```

打开文件时必须确定访问模式，在 PERL 中有三种访问模式：读(<)、写(>)和添加(>>)。缺省的打开方式为读文件。

写文件和添加文件的区别在于写模式将原文件覆盖，原有内容丢失，其形式为：

```
open(outfile,">outfile");
```

而添加模式则在原来文件的末尾处继续添加内容，形式为：

```
open(appendfile, ">>appendfile")。
```

注意，不能对文件同时进行读和写或添加操作。

下面是一些具体的例子：

```
open(SESAME, "filename")       # read from existing file
open(SESAME, "<filename")       # (same thing, explicitly)
open(SESAME, ">filename")       # create file and write to it
open(SESAME, ">>filename")      # append to existing file
```

open 函数的返回值可用来确定打开文件的操作是否成功：成功时返回非零值，失败时返回零，因此可以作出如下判断：

```
if (open(MYFILE, "myfile"))
{      # 文件打开成功所进行的工作
       statements;
}
#文件打开失败时结束程序
```

也可以用逻辑或操作符表示如下：

```
open (MYFILE, "file1") || die ("Could not open file");
```

当文件操作完毕后，可用以下表达式关闭文件。

```
close(MYFILE);
```

下面是一个打开文件并进行操作的例子：

**程序清单 11-1：打开一个文件并进行操作**

```
open(GRADES, "grades") or die "Can't open grades: $!\n";
while ($line = <GRADES>) {
    ($student, $grade) = split(" ", $line);
    $grades{$student} .= $grade . " ";
}
foreach $student (sort keys %grades) {
    $scores = 0;
    $total = 0;
    @grades = split(" ", $grades{$student});
    foreach $grade (@grades) {
        $total += $grade;
        $scores++;
    }
    $average = $total / $scores;
```

```
    print "$student: $grades{$student}\tAverage: $average\n";
}
```

对文件的操作都离不开文件句柄，以下是几个常用的特殊全局文件句柄：

- STDERR —— Perl 启动时打开的缺省文件句柄。STDERR（标准错误）通常初始化为计算机的监视器。
- STDIN —— Perl 启动时打开的缺省文件句柄。STDIN（标准输入）通常初始化为计算机的键盘。从输入变元数组@ARGV 读取程序输入时要用到这个文件句柄。
- STDOUT —— Perl 启动时打开的缺省文件句柄。STDOUT（标准输出）通常初始化为计算机的监视器。

下面是 STDIN 的一种最常用的使用方法：

```
while ($inputLine = <STDIN>) {
    print($inputLine);
}
```

### 11.1.2 读/写文件

1. 读文件

读取文件就是从文件中获得数据，它的形式为：

```
$line = <MYFILE>;
```

这条语句含义是从文件中读取一行数据存储到简单变量$line 中，并把文件指针向后移动一行。又如语句：

```
@array = <MYFILE>;
```

表示把文件的全部内容读入数组@array 中，文件的每一行（含回车符）为数组@array 的一个元素。

2. 写文件

写文件就是把数据写入到指定的文件中，它的形式为：

```
open(OUTFILE, ">outfile");
print OUTFILE ("Here is an output line.\n");
```

### 11.1.3　删除文件

Perl 的 unlink 函数会删除某文件的一个名字（它可能还有其他名称）。当文件的最后一个名称被删除，而且没有进程正在使用时，文件本身会被删除。这是 UNIX rm 命令的实际动作。一般来说，文件通常只有一个名称，所以可以把删除名称直接想象成删除文件。

请看下面的例子：

**程序清单 11-2：删除文件操作**

```
print  "which file do you want to delete?";
chomp   ($name = <STDIN>);
unlink ($name);
```

unlink 函数还可以接受文件名列表：

```
unlink ( "file1" , "file2" );
unlink <*.o>;
```

unlink 函数会返回成功删除的文件数目。如果只有一个参数，而且顺利被删除，返回值就为 1，否则为 0。如果传入 3 个文件名，但只删除了 2 个，那么返回值就是 2。

如果不传参数给 unlink 函数，它会用默认的参数——$_变量。下面是一个例子：

**程序清单 11-3：$_变量在 unlink 参数中的使用**

```
foreach   (<*.o>) {     # 循序读取扩展名为.o的文件
        unlink   ||   warn "Can not delete $_: $!";      # 删除文件，不成功则警告
}
```

### 11.1.4　更改文件名

在 Perl 里，用 rename 函数可完成更改文件名的工作。其使用的语法如下：

```
rename ($old,$new);
```

下面这行语句会把文件 fileold 改名为 filenew：

```
rename ("fileold","filenew");
```

和大多数函数一样，rename 函数如果执行成功，会返回一个数值，可以通过检查返回值，看看 rename 是否成功执行。

rename 函数还可以指定新文件所在的目录，如：

```
rename ("fileold","someDirectory/filenew");
```

下面的程序示意了打开/关闭文件和修改文件名的操作方法：

**程序清单 11-4：修改文件名**

```
open(OLD, "< $old")   or die "can't open $old: $!";
open(NEW, "> $new")   or die "can't open $new: $!";
while (<OLD>) {
    # change $_, then...
    print NEW $_   or die "can't write $new: $!";
}
close(OLD)   or die "can't close $old: $!";
close(NEW)   or die "can't close $new: $!";
rename($old, "$old.orig")   or die "can't rename $old to $old.orig: $!";
rename($new, $old)   or die "can't rename $new to $old: $!";
```

### 11.1.5 产生/删除文件链接

有时候，同一个文件需要一个以上的名称。产生其他名称的动作称为链接（linking）。链接共分为两种，分别是硬链接（hard link）和符号链接（symbolic link）并非每个操作系统都支持以上两种链接。本节描述的是 POSIX 下的文件系统。

1. 硬链接

文件的硬链接和其原始的文件名称基本上是无法分辨的，事实上它等于是“文件的真实姓名”。

操作系统会追踪某个特定时刻参照到同一个文件的 hard link 总数。文件第一次产生时，就有一个链接随之产生。每次新增一个 hard link 时，计数会增加 1；删除时则会减少 1。等到最后一个参考到该文件的链接也消失后，文件内容就会被删除，此后再也参考不到了。

每个参考到同一文件的 hard link 必须位于同一个已挂上的（mounted）文件系统中，这通常是指一个磁盘驱动器或其中的一部分。因此，不能在另一个文件系统中产生某一文件的新的 hard link。

大部分系统对目录的 hard link 有所限制。为了保持完整的树状目录结构，所有的于目录

在根目录下只能有一个名称。如果读者试图替目录产生新的硬链接，会得到错误信息——除非你是超级用户。不过，即使你是超级用户，也不要那么做，那样文件系统将会很乱。

2. 符号链接

符号链接（symbolic link）又称为 symlink 或软链接（soft link）。它是一种特别的文件，其数据是路径名称。打开这个文件时，操作系统会用其中的路径打开位于该处的文件。

举例来说，如果有一个名为 fred 的符号链接，里面有个名字叫 barney 的文件，打开 fred 其实就等于打开 barney。如果 barney 是目录，则 fred / wilma 会参照到 barney / wilma。

symlink 的内容（symlink 所指处）不一定要参考到某个存在的文件或目录。也就是说，产生 fred 时，barney 不需要存在，甚至可以永远不存在。symlink 可以离开目前的文件系统，所以我们可以产生指向其他文件系统的 symlink。此外，symlink 可以指向另一个 symlink，事实上，许多系统至少支持 8 层的 symlink，但实际上很少用到这么多。



Hard link 可以保护文件的内容，防止文件遗失，symlink 却无法做到，但 symlink 可以跨越文件系统，hard link 则不行。目录只能用 symlink。

3. 利用 Perl 产生链接和读取链接信息

在 Perl 中，要产生 hard link，可以这样表示：

```
link ("fred","barney");
```

link 函数需要两个参数，分别是原始文件名称和新的别名，如果成功产生链接，会返回真。

对 hard link 来说，原始文件不能是目录，而新的别名必须位于同一个文件系统内。这些限制正是当初发明 symlink 的理由。

在 Perl 中，可以用 symlink 函数来产生符号链接，下面是一个例子：

```
symlink ("barney","neighbor");
```

这里 barney 不一定要存在，这种情况下，参照 neighbor 会得到“No such file or directory”的错误信息。

Perl 提供了 readlink 函数来读取链接信息，其调用语法如下：

```
filename = readlink (linkname);
```

如果 linkname 为符号链接文件，则返回其实际指向的文件；否则返回空串。

4. 删除文件链接

Perl 中删除文件链接的函数为 unlink，其调用语法为：

```
num = unlink (filelist);
```

该函数的参数为文件名列表，返回值为实际删除的文件数目。它之所以叫 unlink 而不叫 delete，这是因为它实际所做的是删除文件的链接，而不是文件。

### 11.1.6 判断文件状态

1. 文件测试操作符

下面这个例子的 if 语句中的“–e”1 就是测试文件是否存在的操作符：

```
if (-e "/path/file1") {
    print STDERR ("File file1 exists.\n");
}
```

文件测试操作符的语法为：

```
-op expr
```

表 11-1 列出了各种文件测试操作符：

表 11-1　　文件测试操作符

| 操 作 符 | 描　　述 |
|---|---|
| -b | 是否为块设备 |
| -c | 是否为字符设备 |
| -d | 是否为目录 |
| -e | 是否存在 |
| -f | 是否为普通文件 |
| -g | 是否设置了 setgid 位 |
| -k | 是否设置了 sticky 位 |
| -l | 是否为符号链接 |
| -o | 是否拥有该文件 |

续表

| 操 作 符 | 描　　述 |
| --- | --- |
| -p | 是否为管道 |
| -r | 是否可读 |
| -s | 是否非空 |
| -t | 是否表示终端 |
| -u | 是否设置了 setuid 位 |
| -w | 是否可写 |
| -x | 是否可执行 |
| -z | 是否为空文件 |
| -A | 距上次访问时间 |
| -B | 是否为二进制文件 |
| -C | 距上次访问文件的 inode 多长时间 |
| -M | 距上次修改时间 |
| -O | 是否只为“真正的用户”所拥有 |
| -R | 是否只有“真正的用户”可读 |
| -S | 是否为 socket |
| -T | 是否为文本文件 |
| -W | 是否只有“真正的用户”可写 |
| -X | 是否只有“真正的用户”执行 |



“真正用户”指登录时指定的 userid，与当前进程用户 ID 相对，命令 suid 可以改变有效用户的 ID.

下面是另一个具体的例子：

**程序清单 11-5：文件测试操作符的使用**

```
unless (open(INFILE, "infile")) {
    die ("Input file infile cannot be opened.\n");
}
if (-e "outfile") {
```

```
    die ("Output file outfile already exists.\n");
}
unless (open(OUTFILE, ">outfile"))  {
die ("Output file outfile cannot be opened.\n");
}
foreach (@ARGV) {
    print;
    print((-f) ? " -REGULAR\n" : " -SPECIAL\n")
}
```

程序执行情况如下：

```
perl -w 09lst01.pl \perl5 perl.exe \windows      # 输入命令
09lst01.pl -REGULAR
\perl5 -SPECIAL
perl.exe -REGULAR
\windows -SPECIAL
```

2. 取得文件状态的函数

Perl 提供的 stat 函数可以获取文件的当前状态。其调用语法如下：

```
stat (file);
```

参数 file 可为文件名也可为文件变量。

该函数返回的列表元素依次为：

- 文件所在设备；
- 内部参考号(inode)；
- 访问权限；
- 硬链接数；
- 属主的(数字)ID；
- 所属组的(数字)ID；
- 设备类型（如果 file 是设备的话）；
- 文件大小（字节数）；
- 最后访问时间；
- 最后修改时间或最后改变状态时间；
- I/O 操作最佳块大小；
- 分配给该文件的块数。

另外还有一个函数——lstat，其调用语法为：

```
lstat (file);
```

该函数与 stat 函数类似，区别是将 file 看作是符号链接。

### 11.1.7　读取文件中的某一行

下面的函数可以读取文件中的某一指定的行：

**程序清单 11-6：读取文件中的某一指定行的函数**

```
sub build_index {
    my $data_file   = shift;
    my $index_file = shift;
    my $offset = 0;
    while (<$data_file>) {
        print $index_file pack("N", $offset);
        $offset = tell($data_file);
    }
}
sub line_with_index {
    my $data_file    = shift;
    my $index_file   = shift;
    my $line_number = shift;
    my $size;      # size of an index entry
    my $i_offset;      # offset into the index of the entry
    my $entry;      # index entry
    my $d_offset;      # offset into the data file
    $size = length(pack("N", 0));
    $i_offset = $size * ($line_number-1);
    seek($index_file, $i_offset, 0) or return;
    read($index_file, $entry, $size);
    $d_offset = unpack("N", $entry);
    seek($data_file, $d_offset, 0);
    return scalar(<$data_file>);
}
open(FILE, "< $file")   or die "Can't open $file for reading: $!\n";
open(INDEX, "+>$file.idx")   or die "Can't open $file.idx for read/write: $!\n";
build_index(*FILE, *INDEX);
$line = line_with_index(*FILE, *INDEX, $seeking);
```

# 11.2 目录操作

上节介绍了对文件的一些操作，本节将介绍对目录的操作。

## 11.2.1 产生/删除目录

1. 产生目录

UNIX 有一个 mkdir 命令可以用来产生目录，Perl 也有一个同名的函数。Perl 的 mkdir 函数需要两个参数，第一是新目录的名称，第二是该目录的权限。

该函数的调用语法如下：

```
mkdir (dirname, permissions);
```

创建新目录，它的两个参数的意义为：

- dirname——将要创建的目录名，可以为字符串或表达式。
- permissions——八进制数，指定目录的访问权限。它的值和意义见表 11-2，权限的组合方法为将相应的值相加。

表 11-2 创建目录的权限

| 值 | 权　限 |
|---|---|
| 4000 | 运行时设置用户 ID |
| 2000 | 运行时设置组 ID |
| 1000 | 粘贴位 |
| 0400 | 拥有者读权限 |
| 0200 | 拥有者写权限 |
| 0100 | 拥有者执行权限 |
| 0040 | 组读权限 |
| 0020 | 组写权限 |
| 0010 | 组执行权限 |
| 0004 | 所有人读权限 |
| 0002 | 所有人写权限 |
| 0001 | 所有人执行权限 |

2. 删除目录

Perl 采用 rmdir 函数来删除空的目录，其调用语法如下：

```
rmdir (dirname);
```

该函数执行成功则返回非零值，失败返回零值。

### 11.2.2　打开/关闭目录

1. 打开目录

打开目录的函数为 opendir，其调用语法如下：

```
opendir (dirvar, dirname);
```

该函数与下面将要介绍的几个函数合用，可查看某目录中的文件列表。它的两个参数的意义为：

- dirvar——目录变量，与文件变量类似。
- dirname——目录名，可为字符串或表达式，该函数成功则返回真值，失败则返回假。



程序中可用同名的目录变量和文件变量，根据环境确定其成分。

2. 关闭目录

Perl 使用 closedir 函数来关闭已打开的目录，其调用语法如下：

```
closedir (mydir);
```

### 11.2.3　其他目录处理函数

1. 改变当前工作目录函数

chdir 为改变当前工作目录函数，其调用语法为：

```
chdir (dirname);
```

参数 dirname 可以为字符串，也可以为表达式。

下面的程序首先将目录改变，然后打开该目录下的 Foo 个文件，赋给文件句柄 FOO，然后读取文件中的数据，最后关闭该文件。

**程序清单 11-7：改变当前工作目录**

```
chdir $dir   or die "chdir $dir: $!";
open FOO, $file   or die "open $file: $!";
@lines = <FOO>   or die "$file is empty?";
close FOO   or die "close $file: $!";
```

2. 读取目录函数

readdir 为读取目录函数，其调用语法为：

```
readdir (mydir);
```

当赋值给简单变量时，每次赋予一个文件或子目录名，对数组则赋予全部文件和子目录名。

下面是一个使用读取目录函数的例子：

```
opendir(DIR, $dirname) or die "can't opendir $dirname: $!";
while (defined($file = readdir(DIR))) {
    # do something with "$dirname/$file"
}
closedir(DIR);
```

从上面的例子中我们可以看到，要对目录进行读取操作，必须先打开目录，操作完之后，还必须关掉目录。

下面的程序可以得到指定目录中所有匹配指定模式的文件名：

**程序清单 11-8：得到指定目录中的匹配的文件名**

```
opendir(DIR, $path);
@files = grep { /\.c$/ } readdir(DIR);
closedir(DIR);
```

3. 与目录位置有关的函数

telldir 为取得目录位置的函数，其调用语法为：

```
location = telldir (mydir);
```

像在文件中前后移动一样，telldir 和下面的 seekdir 用于在目录列表中前后移动，前者得到想要的目录位置，后者将目录移到指定的位置。seekdir 函数的调用语法为：

```
seekdir(mydir, location);
```

其参数 location 必须为 telldir 返回的值。rewinddir 函数将读取目录的位置重置回开头，从而可以重读目录列表，其调用语法为：

```
rewinddir (mydir);
```

## 11.3　更改文件和目录的权限、所有权和时间戳

### 11.3.1　更改权限

文件或目录的权限定义了“谁”能对它们做“什么”事。在 UNIX 下，改变权限的传统方法是使用 chmod 命令。在 Perl 中，仍然使用 chmod 函数。其调用语法如下：

```
chmod (permissions, filelist);
```

该函数可改变文件的访问权限。其两个参数的含义为：
permissions 为将要设置的访问权限，其含义见本章第 2 节中 mkdir 的权限表。
filelist 为欲改变权限的文件列表。
此函数的参数是一个八进制数字和一堆文件名，数字代表权限设定，而该函数会试图把这些文件的权限改成数字代表的设定。举例来说，下面的程序会让所有人都能读写 fred 和 barney 这两个文件：

```
chmod (0666, "fred" ,"barney");
```

函数的返回值代表成功改变权限的文件数，即使新设定和原设定相同也算。

(group)。文件的所有者和群组决定了要对谁做怎样的权限限制。文件的所有者和群组是在文件产生时决定的。但是在某些情况下，你可以更改这些值。实际的情况每个系统都不一样，具体请参考改变文件的宿主函数 chown 的文档。chown 函数的调用语法为：

```
chown (userid, groupid, filelist);
```

该函数有三个参数，它们的含义分别为：

userid：新属主的(数字)ID 号。

groupid：新的组(数字)ID 号，-1 为保留原组。

filelist：欲改变宿主的文件列表。

chown 函数接受 UID、GID 以及一连串文件名称做为参数，并试图更改这些文件的所有权。如果成功，返回值为代表成功更动的文件数，这和 chmod 及 unlink 等函数一样。注意这样做会同时改动所有者和群组。如果不想改变群组，可以在 GID 参数字段写入-1。还有一点要注意的事，无论是 UID 或 GID 都要用数字表示，不能用对应的文字名称。举例来说，如果 fred 的 UID 是 1234，其默认群组 stoners 的 GID 是 35，下面这个例子会让文件 file1 和 file2 属于 fred 和其默认群组所有：

```
chown(1234，35，"file1","file2");
```

### 11.3.3 更改时间戳

每个文件都有对应的三个时间戳（timestamp）。这些时间戳分别代表三种和文件有关的时间信息：

- 最后一次存取时间。
- 最后一次更改时间。
- 最后一次 inode 改变的时间。

前两个时间戳可以用 utime 函数改变其中的内容。设定前两个时间戳之值会自动把第三个时间戳改成目前的时间，所以 Perl 不提供（其实也不需要）设定第三种时间戳的方法。时间戳的值以内部时间格式储存，即从 GMT 时间 1970 年 1 月 1 日午夜开始，以秒为单位计算的整数。系统内部用一个 32 位无符号数储存这个数字。

utime 函数使用的方式和 chmod 及 unlink 函数很像。其调用语法为：

```
utime (acctime, modtime, filelist);
```

下面是一个例子：

**程序清单 11-9：更改时间戳函数**

```
$acctime = -A "file1";
$modtime = -M "file1";
@filelist = ("file2", "file3");
utime ($acctime, $modtime, @filelist);
```

该函数以文件名列表做为参数，以返回有效作用的文件数。

时间戳的数字可以随便设定，你甚至可以让文件“看起来”在一段时间之后才会被改变。下面的例子会让文件 someFile“看起来”在 50 分钟后被改变：

```
$when = time() + 50*60;
utime ($when，$when，"someFile");
```

上面的例子中用到了 time()函数，它返回从 1970 年 1 月 1 日起累计秒数，调用语法为：

```
currtime = time();
```

另外还有几个与时间有关的函数，分别介绍如下：

1. gmtime 函数

该函数将由 time，stat 或 -A 和 -M 文件测试操作符返回的时间转换成格林威治时间。其调用语法为：

```
timelist = gmtime (timeval);
```

该函数返回的列表的元素依次为：

秒
分
小时，0~23
日期
月份，0~11(1 月~12 月)
年份
星期，0~6(周日~周六)
一年中的日期，0~364
是否夏令时的标志

2. localtime 函数

该函数与 gmtime 类似，区别是它将格林威治标准时间值转换为本地时间。该函数的调用语法为：

```
timelist = localtime (timeval);
```

详细信息请参阅有关手册。

# 11.4 使用 DBM 文件

所有 UNIX 系统都会有一套称为 DBM 的标准函数库。这个函数库提供有简单的数据库存取功能，供应用程序将 key-value 序对存入磁盘中。应用程序可以读取、写入或删除这些数据。

Perl 存取 DBM 数据库的方式很简单。可以先将一个散列变量（又称为 DBM 数组）和 DBM 数据库关联起来。随后，这个 DBM 数组就可以用来存取和更改数据库的内容，在数组中产生新元素，表明在 DBM 数据库中产生新元素，删除也类似。

DBM 数据库中 key 和值的大小、数目以及种类都有所限制。在某些 DBM 函数库版本中，DBM 数组也会有这些限制。一般来说，如果 key 和值都小于 1000 个二进制字符，应该不会有什么问题。

## 11.4.1 打开/关闭 DBM 文件

让 DBM 数据库和 DBM 数组建立关联很简单，只要利用 dbmopen 函数就可以了。其调用语法如下：

```
dbmopen (array, dbmfilename, permissions);
```

各个参数的意义为：

array：所用的关联数组的名称。

dbmfilename：将打开的 DBM 文件名。

permissions：访问权限，详细情况请查看表 11-2。

这个散列内部的值会被清掉，随后和名称为 array 的 DBM 数据库产生关联；数据库在磁盘中通常会以 dbmfilename.dir 和 dbmfilename.pag 成对文件的形式存在。

参数 permissions 代表权限的控制位，通常用 8 字节的方式表示，常用的设定是 0644。代表只有所有者同时有读、写权限，其他人都只有读取权。如果文件本来就存在，此参数没有作用。

请看下面的例子：

```
dbmopen(%fred, "myDataBase",0644);
```

上面这行语句打开以%fred 表示的 myDataBase 文件，它会把散列变量%fred 与当前目录中名为 myDataBase.dir 和 myDataBase.pag 的文件关连起来。如果文件不存在，会用 0644 的设定，配合目前的 umask（该函数的作用为设置文件访问权限的掩码，其用法为：oldmaskval = umask (maskval)；返回值为当前掩码）设定值产生新文件。

若文件打开成功，dbmopen 会返回真，否则为假，这和 open 函数一样。DBM 数组在整个程序中都会维持打开的状态。程序结束时，关联状态也就结束。当然，你可以像关闭文件

的设定，配合目前的 umask（该函数的作用为设置文件访问权限的掩码，其用法为：oldmaskval = umask (maskval)；返回值为当前掩码）设定值产生新文件。

若文件打开成功，dbmopen 会返回真，否则为假，这和 open 函数一样。DBM 数组在整个程序中都会维持打开的状态。程序结束时，关联状态也就结束。当然，你可以像关闭文件句柄一样，用 dbmclose 函数提前结束关联状态，其调用语法如下：

```
dbmclose (array);      # 关闭DBM文件，拆除关联数组与之的关系。
```

该函数和 close 一样，如果有问题就会返回假值。

### 11.4.2　DBM 散列的使用

前面已经讲过，数据库打开之后，对 DBM 散列变量的任何存取动作，都会对应到数据库。对前面的那个%fred 来说，我们可以用它来新增、删除或存取数据库中的元素：

**程序清单 11-10：DBM 散列的使用**

```
%fred ("fruit1") = "apple";     # 建立或更新一个元素
delete $fred {"fruit1"}       # 删除key值为 " fruit1 " 的元素
$key   (keys %fred)   {       # 扫过所有的值
    print "$keys has value of $fred{$key} \n";
}
```

最后面的循环会把整个文件扫过两次：第一次存取所有的 key，第二次查找这些 key 所对应的值。如果我们使用 each 操作符，只要读取一次即可，所以效率更高。其语法结构如下所示：

```
while (($key , $value) = each (%fred)) {
    print    " $keys has value of $fred{$key} \n " ;
}
```

## 11.5　小结与练习

### 11.5.1　小结

本章重点介绍了文件系统。对文件的操作是每一种编程语言都具备的功能。

本章第 1 节介绍了文件操作，即如何打开/关闭文件、读/写文件、删除文件以及更改文件名等操作，这些操作都有相应的函数与之对应。然后介绍了产生和删除文件链接，链接中

第 3 节是关于如何更改文件和目录的权限、所有权和时间戳的。文件或目录的权限定义了“谁”能对它们做“什么”事。在 UNIX 下，目录和文件都是具有相应的权限及所有者的。每个文件都有对应的三个时间戳，这些时间戳可以通过函数来改变。

第 4 节介绍了如何使用 DBM。DBM 是所有 UNIX 系统都具备的一套标准函数库。这个函数库提供了简单的数据库存取功能，供应用程序将 key-value 序对存入磁盘中。应用程序可以读取、写入或删除这些数据。

学习了本章，读者应该掌握 Perl 中的文件系统。下一章将介绍输入/输出处理。

### 11.5.2 习题与思考

（1）写一个程序，从标准输入设备 STDIN 中读取文件名,然后打开该文件，显示每一行的内容，并在内容之前写上文件名。比如一个叫 fred 的文件当中包含三行内容：aaa, bbb 和 ccc, 那么输出应该是 fred: aaa, fred: bbb 和 fred: ccc。

（2）写一个程序，输入一个输入文件名、一个输出文件名，一个查找模式和一个替换字符串。然后将输入文件中所有和查找模式相匹配的字符串用替换字符串代替后，将输入文件复制到输出文件中。

（3）写一个程序，读入一串文件名，然后判断哪个文件是可读的、可写的和不存在的。

（4）写一个程序，读入一串文件名，然后判断哪个文件创建的时间最早，输出它的文件名和它存在的时间。

# 第12章

## 输入/输出处理

基本输入
基本输出
格式化输出
小结与练习



本章将介绍与输入输出有关的一些问题。包括基本输入、基本输出函数以及格式化输出的基本方法。首先介绍了大家非常熟悉的 print 函数及其语法。如果 print 函数格式化输出数据的方法不适合你的要求，Perl 还提供了格式化报表输出的特殊语法。如果要分栏打印大量数据，可以用 Perl 5 的 write 函数和 format 语句。

本章要点：

- print 函数的使用。
- printf 函数和 sprintf 函数的使用及其区别。
- 格式化输出方法的使用。
- 几种字段容器的使用。

# 12.1 基本输入

## 12.1.1 来自 STDIN 的输入

前面读者已经见到过使用<STDIN>读取输入数据的例子了，这里再做一下详细的讨论。

在 Perl 程序中，<STDIN>的作用是由标准输入设备读取数据。在标量环境下，该语句会返回输入的下一行；如果已无数据，则会返回 undef。下面就是一个例子：

```
$x = <STDIN>;
```

该语句在数组环境下会返回一个列表，内容是剩下的所有数据，其中每个元素由一行构成（包括换行符）。请看下面的例子：

```
@x = <STDIN>;
```

我们常常利用输入语句控制循环的进行，若循环测试仅由输入语句构成，Perl 会自动把读入的数据存放在变量$_中。

**程序清单 12-1：来自 STDIN 的输入**

```
while(<STDIN>)     # 同：while (defined ($_ = <STDIN>))
{
    chomp;     # 同：chomp($_);
    print $_,"\n";
}
```

许多函数或操作符都以$_做为默认操作符，读者只要稍微用点心，就能写出简洁易懂的

程序。

### 12.1.2　输入操作符

另外还有一个办法可以读取输入，那就是使用输入操作符“<>”。输入操作符的用法和<STDIN>类似，唯一不同的地方是，输入操作符会由文件（一个或一个以上）读入数据，而这些文件的名字是在程序执行时，由命令行的参数来决定的。例如，有个名为 hello 的程序内容如下：

**程序清单 12-2：输入操作符的使用**

```
while(<>) {
    print ;
}
```

然后我们执行

```
hello   file1   file2   file3
```

此时，hello 会一行行地读入 file1 的内容，接着是 flle2、file3，直到 file3 的最后一行也读完了，<>返回 undef，跳出循环，让程序结束。

严格地说来，输入操作符其实不会去检查命令行的参数；它检查的是一个名为@ARGV 的数组，Perl 解释器会把它的内容设成命令行的参数。虽然如此，该数组的内容还是可以改变的。

### 12.1.3　@ARGV 数组

现在再来讨论一下前面提到的@ARGV 数组。读者如果以前曾经使用过 C 语言，可能会见过它，它就是 main()函数的一个参数。利用@ARGV 特殊变量可以直接从命令行取得信息，而不需要特殊语句。Perl 5 自动取得命令行中的 Perl 程序名之后和按 Enter 键之前的所有项目并使用它来生成数组@ARGV。

@ARGV 数组是一个标准的 Perl 5 数组，它是由 Perl 5 解释器在启动时生成的。Perl 5 中的@ARGV 数组与 C 语言中的 argv 和 argc 变量不同，它并不包含调用程序名。Perl 5 将调用程序名放在变量$0 中，这一点可以由下面的例子来说明：

```
$var = $ARGV[0];    # 第一个参数
$numargs = @ARGV;    # 参数的个数
```

@ARGV 数组的每个单元包含命令行中的一个空格分隔项目。可以在装入信息时将输入

字符串放在引号中。如果有一个名为 argv.pl 的 Perl 5 程序，调用如下：

```
argv.pl  context.pl  env.pl  "fibonacci  Rev2.pl"  fileInfo.pl
```

则@ARGV 数组会包含四个数组元素（注意 Perl 5 中的@ARGV 数组并不包含调用程序名）。这四个元素分别为：

```
$ARGV[0] = context.pl;
$ARGV[1] = env.pl;
$ARGV[2] = fibonacci  Rev2.pl;
$ARGV[0] = fileInfo.pl;
```

此例中如果文件名"fibonacci  Rev2.pl"不放在引号中，则会使它变成两个数组，即单元 2 包含 fibonacci ，单元 3 包含 Rev2.pl，而这两者都不表示实际的文件，显然是行不通的。

除了@ARGV 数组外，还生成了下列特殊变量：

$#ARGV 保存@ARGV 数组中的元数个数。

ARGV 包含程序用@ARGV 数组打开文件时的文件句柄名。

$ARGV 包含与 ARGV 数组文件句柄相关联的文件句柄。



文件句柄是计算机程序与磁盘文件之间的直接链接，它用于读取和写入文件。

在 Perl 中，<>操作符实际上是对数组@ARGV 的隐含引用，其工作原理如下：

第一步：当Perl解释器第一次看到<>时，打开以$ARGV[0]为文件名的文件。

第二步：执行动作shift(@ARGV)，即把数组@ARGV的元素向前移动一个，其元素数量即减少了一个。

第三步：<>操作符读取在第一步打开的文件中的所有行。

第四步：读完后，解释器回到第一步重复。

下面是一个例子：

**程序清单 12-3：@ARGV 变量的使用**

```
@ARGV = ("myfile1", "myfile2");    # 实际上由命令行参数赋值
while ($line = <>) {
    print ($line);
}
```

该程序的执行结果是把文件myfile1和myfile2的内容打印出来。

## 12.2　基本输出

Perl 利用 print 和 printf 这两个函数，对标准输出设备做写入的动作。

### 12.2.1　print 函数

print 函数虽然我们已经使用了很多次，但是这里还是要再专门讨论一下。该函数可接受一个由字符串构成的列表作为参数，并逐一把列表中的字符串输出到标准输出设备上，不对其内容做任何变动。执行成功会返回真，否则为假。

请看下面的例子：

```
$x = print ( " How " , " are " , " you " , " \n " );
```

上面的语句执行成功后，在屏幕上打印出“How are you”，并使$x = 1。

虽然说参数的括号可加可不加，不过如果想打印出的第一个元素加上括号，就必须把整个参数都括起来才行。请看下面的例子：

**程序清单 12-4：print 函数的使用**

```
print   1+2, "hello","\n";
print   (1+2),"hello","\n";
print   ((1+2), "hello","\n");
```

程序打印结果如下：

```
3hello
33hello
```

从这个结果中可以看出，第二行程序仅仅输出了一个 3，而忽略了后面的“hello”和回车，就是因为第二行中的第一个参数使用了括号。

### 12.2.2　格式化输出函数 sprintf 和 printf

如果读者使用过 C 语言编写程序，一定会对功能强大的 printf 函数记忆深刻，Perl 也提供类似的同名函数。

sprintf 和 printf 函数的作用基本相同，都是根据格式代码产生格式化输出。sprintf 函数通

常用于对一个或几个变量采用特定格式，然后可以在程序其他地方使用这些变量。printf 函数通常用于打印某些其中变量需要特定格式的字符串。

1. sprintf 函数

sprintf 函数返回字符串中的已经格式化的变量。其常见的用法如下：

```
$formattedVariable = sprintf  " %formatcode " ,$unformattdVariable;
```

格式代码是可选的。如果指定格式代码，则 sprintf 函数将其用于$unformattedVariable，产生的格式化字符串保存在$formattedVariable 中，格式代码前面带%号。

2. printf 函数

printf 函数将格式化输出打印到指定的文件句柄。其通常用法如下：

```
printf  (FILEHANDLE  " format " ,formatVariableList);
```

printf 函数有以下 3 个参数：

- 输出文件句柄，可选。
- 格式字符串，可以包含格式代码。
- 要格式化的变量。

3. sprintf 和 printf 函数的数据格式

sprintf 和 printf 函数的数据代码都可以包括格式标志、宽度指定符和数据格式化代码。可以用 sprintf 和 printf 函数的格式标志指定变量的对齐方式和所用符号类型。宽度指定符是标志后面的数字，表示格式化中使用的空格数。宽度指定符数字可以分为两个部分：总字段长度和小数位个数，两个部分用句号分开，如下所示：

```
total-field-size.number-of-decimal-places
```

用总字段长度减去小数位数的长度再减去 1（小数点），就可以得到整数值或科学计数的空格数。表 12-1 列出了格式标志和宽度指定符的例子。

表 12-1　　sprintf 和 printf 函数的格式代码

| 标　志 | 描　述 | 例　子 |
| --- | --- | --- |
| 空（缺省） | 正数以空格开头，负数以负号开头，左边填充空格 | $Y = printf " %　4.2\n " ,$X; |

续表

| 标　志 | 描　述 | 例　子 |
| --- | --- | --- |
| - | 正数以空格开头，负数以负号开头，左对齐字符，右边填充空格 | $Y = printf " %-4.2\n " ,$X; |
| + | 正数以正号开头，负数以负号开头，右对齐字符，左边填充空格 | $Y = printf " %+4.2\n " ,$X; |

数据类型格式代码告诉 Perl 5 要显示哪种数据格式。可以强制 Perl 5 显示整数、十进制数、十六进制数、字符、科学计数和其他格式，具体见表 12-2。

表 12-2　　sprintf 和 printf 函数的数据格式

| 格　式 | 描　述 |
| --- | --- |
| c | 字符格式 |
| d | 十进制格式 |
| e | 指数格式 |
| f | 浮点数格式 |
| g | 紧缩形式浮点数 |
| ld | 长十进制格式 |
| lo | 长八进制格式 |
| lu | 长无符号十进制格式 |
| lx | 长十六进制格式 |
| O | 八进制格式 |
| S | 字符串格式 |
| U | 无符号十进制格式 |
| X | 十六进制格式 |
| X | 大写十六进制格式 |

下面就是一个使用 sprintf 函数格式化输出的例子：

**程序清单 12-5：sprintf 函数的使用**

```
$a = 0.255;
$b = sprintf("%.2f", $a);
print "Unrounded: $a\nRounded: $b\n";
printf "Unrounded: $a\nRounded: %.2f\n", $a;
```

该程序输出的结果为：

```
Unrounded: 0.255
Rounded: 0.26
Unrounded: 0.255
Rounded: 0.26
```

下面的程序也是一个使用 printf 函数格式化输出的例子，其中使用了 POSIX 模块，因为程序中要使用的 floor 和 ceil 函数在该模块中。

**程序清单 12-6：printf 函数的使用**

```
use POSIX;
print "number\tint\tfloor\tceil\n";
@a = ( 3.3 , 3.5 , 3.7, -3.3 );
foreach (@a) {
    printf( "%.1f\t%.1f\t%.1f\t%.1f\n",
        $_, int($_), floor($_), ceil($_) );
}
```

程序输出结果为：

```
number   int      floor    ceil
 3.3      3.0      3.0      4.0
 3.5      3.0      3.0      4.0
 3.7      3.0      3.0      4.0
-3.3     -3.0     -4.0     -3.0
```

## 12.3 格式化输出

### 12.3.1 什么是格式

Perl 的含义为“实用摘录与报告语言”（Practical Extraction and Report Language）。现在我们就来讨论其中的“报告语言”方面的内容。

Perl 提供了一种简单的报告书写模板，我们称之为“格式”（format）。格式定义了许多

常数部分，包括字段标题、标记、固定字样，以及变量部分（即输出数据）应该呈现出的样子。

使用格式包括三个步骤：

（1）定义格式。

（2）把要打印出的数据套进格式的变量部分。

（3）使用格式。

一般来说，只需在原始代码中定义一次格式，而另两个步骤可以重复进行许多次。

### 12.3.2　怎么定义格式

格式的定义可以放在程序代码的任何地方，和函数定义一样。下面是格式定义的一种示例：

```
format formatname =
fieldline
value1,value2,value3
fieldline
value1,value2
fieldline
value1,value2,value3
.
```

第一行最开头是 format 保留字，其后是格式的名称和“=”符号。格式名称有自己的命名空间，命名规则和其他变量的命名一样。格式名称不会在程序里面直接用到，所以就算和关键字同名也不要紧。

紧接在第一行下面的是模板，可能一行都没有，也可能是许多行，模板最后必须用句点“.”做结尾。模板对空白字符是敏感的，一个空白与两个空白或 tab 键意义完全不一样。模板定义包含一连串的字段行 (fieldline)。字段行可以包含固定的文字，它们会原封不动地打印到屏幕上。

字段行里面还可以包含字段容器（fieldholder），用来储存变量文字。若某字段行包含字段容器，应在接下去的那一行（称为值行，valueline）存放变量，字段容器会对应到它们的值。下面是一个例子：

```
Hello, my name is @<<<<<<<<<<
$name
```

@<<<<<<<<<<是字段容器，代表向左对齐、长度为 12 个字符的容器。该格式的输出形式如：

```
Hello , my name is Tom   Jackson.
```

字段行里面如果有一个以上的字段容器，那么值行也要有这么多数目的变量，它们之间必须用逗号隔开，如下所示：

```
Hello,my name is @<<<<<<<<<<< and I graduated from @<<<<<<<<<
$name,$university
```

利用以上的知识，用户可以设计一个简单的地址标签如下：

```
format   ADDRESSLABEL =
********************************
| @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<|
$name
| @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<|
$address
| @<<<<<<<<<<, @<<<<<, @<<<<|
$city,              $state,     $zip
********************************
.
```

注意，格式的第一行没有任何字段容器，所以下面不需要跟着值行。如果在下一行放置变量，Perl 还是会把它当做字段行，因此往往会发生无法预期的结果。值行中的空白会被忽略。有些人为了看起来方便一些，会把值行中的变量和上行对应的字段容器对齐，这样也不会对输出造成影响。

### 12.3.3 使用格式

利用 write 函数使用格式。该函数的参数是文件句柄，它会用目前的格式产生文字，对参数进行写入的动作。默认格式是和文件句柄同名的格式，所以若用 STDOUT 当参数，就会用到 STDOUT 格式。

下面是一个利用上一节定义的地址标签所写的程序：

**程序清单 12-7：格式的使用**

```
format   ADDRESSLABEL =
********************************
| @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<|
$name
| @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<|
```

```
$address
| @<<<<<<<<<<, @<<<<< @<<<<|
$city,              $state,     $zip
********************************

.
open (ADDRESSLABEL,">labels-to-print") || die"can't create";
open (ADDRESSES,"addresses") || die"can't open addresses";
while (<ADDRESS>)
{
    chomp;     # 将换行去掉
    ($name,$address,$city,$state,$zip) = split ( /:/);     # 分隔，载入全局变量
    write (ADDRESSLABEL);     # 输出
}
```

注意，输出文件的文件句柄和格式名称相同，保证这一点是很重要的。接下来，从地址文件中读取数据，并假设数据是类似下面这样存放的：

```
Stonehenge：56783  SW  Hall Suite  107：Beaverton：OR：97005
Tom Jackson：377  Hard  Rock  lane：Bedrock：OZ：999bc
```

也就是说，这个格式是由冒号分隔的五个字段，程序会以下面的方式来解析它：

wille 一行行循环读入数据，chomp 会去删除换行符号，然后用 split 函数切成五份分别设给对应的变量。注意，这里变量的名称和字段中所用的一定要相同，必须保证这一点。随后，write 函数利用先前定义的格式（名称和传入的文件句柄相同），把文字写到输出文件中。

系统会用值行变量当前的值，取代对应位置的字段容器，然后把它打印出来。下面是执行的结果：

```
********************************
| Stonehenge                    |
| 56783  SW  Hall Suite  107    |
| Beaverton  ,OR        97005   |
********************************
********************************
| Tom Jackson                   |
| 377  Hard  Rock  lane         |
| Bedrock    , OZ       999bc   |
********************************
```

实际的数据一般存放在文件里，比如下面的数据存放在文件 FORMAT.DAT 中：

**程序清单 12-8：测试数据**

```
The Lion King!
Tumbleweed Connection!Elton John!123.32
Photographs & Memories!Jim Croce!4.95
Heads & Tales!Harry Chapin!12.50
```

下面我们就对这些数据进行格式化输出：

**程序清单 12-9：格式化输出数据**

```
open(FILE, "<format.dat");
@lines = <FILE>;
close(FILE);
foreach (@lines) {
    chop;
    ($album, $artist, $price) = (split(/!/));
    print("Album=$album    Artist=$artist    Price=$price\n");
}
```

程序输出结果为：

```
Album=The Lion King    Artist=    Price=
Album=Tumbleweed Connection    Artist=Elton John    Price=123.32
Album=Photographs & Memories    Artist=Jim Croce    Price=4.95
Album=Heads & Tales    Artist=Harry Chapin    Price=12.50
```

在这可以看到，格式化输出之后的结果特别清楚，让人一目了然，这也就是为什么要使用格式化输出的理由。

下面这些特殊的文件句柄变量通常与 format 一起使用：

表 12-3　　与 format 相关的特殊的文件句柄变量

| 变 量 | 英 文 名 | 说　明 |
|---|---|---|
| $\| | $OUTPUT_AUTOFLUSH | 设置为非零值，在每次写入或打印到当前所选输出设备后强制刷新（缺省为零） |
| $% | $FORMAT_PAGE_NUMBER | 当前所选输出文件句柄的当前页号 |
| $= | $FORMAT_LINES_PER_PAGE | 当前所选输出文件句柄的当前页长（可打印行数） |

续表

| 变 量 | 英 文 名 | 说　明 |
|---|---|---|
| $- | $FORMAT_LINES_LEFT | 当前所选输出文件句柄的页面中余下的长度 |
| $~ | $FORMAT_NAME | 当前所选输出文件句柄的当前格式名 |
| $^ | $FORMAT_TOP_NAME | 当前所选输出文件句柄的当前页头格式名 |

### 12.3.4　字段容器

前面我们已经知道了字段容器是什么，但是对于具体怎么使用它去还不是很清楚，下面我们就来详细讨论一下以下几种字段容器的使用：

1. 文本字段

大部分字段容器以@开头，@号后面的字符决定了字段的型态，而字符的数目则决定字段的长度。

@号后面的字符若是左尖括号（<），此字段会“向左对齐”。意思是，如果实际字符串比字段长度短，会从字段的左边开始显示，多出来的位置补以空格；若实际字符串超过字段的长度，多出来的部分会被切掉。@号后面的字符若是右尖括号（>）就代表“向右对齐”；如果是管道符（|）则是“居中对齐”。

2. 数字字段

另外还有一种字段容器是固定精确度的数字字段。这种字段也是以@号开头，后面跟着一个或一个以上的#号，中间可能有句点（代表小数点）。注意@号在这里也计入字段的长度。下面就是一个例子：

```
format  MONEY =
Assets: @#####.##  Liabilities: @#####.##  Net: @#####.##
$assets, $liabilities, $assets - $liabilities
```

这三个数字字段的格式是一样的，容许六位整数（包括@号在内）和两位小数，像这样的格式定义在日常生活中相当有用。

3. 多行字段

前面我们曾经提过，Perl 在正常情况下直接输出结果时，会在遇到的第一个换行符号处停止。解决的方法是利用多行字段 (multiline fields)，它可以用来储存同时包含多行数据的值。多行字段的表示法是@*，它独占一行，下面的行则定义用来取代字段的值。这些替代值打印到屏幕上时会分行出现。下面是一个例子：

程序清单 12-10：多行字段的使用

```
format  STDOUT =
Text  Before.
@*
$longString
Text  after.
.
$longString =  " apple\nbanana\npear\npeach\n " ;
write;
```

程序执行的输出结果为：

```
Text  Before.
apple
banana
pear
peach
Text  After.
```

4. 填满字段

填满字段（filled fields）允许程序员制造填满的段落，将输出文字根据边界来分行，或是根据需要断行等等。这个工作包括几项要点，下面是详细的分析。

填满字段的第一个字符以“^”号表示。填满字段的对应值必须是用来存储字符串的标量变量，不能是传回标量值的语句，这是因为，Perl 会在填满字段时改动变量。这一点和前面不同，请特别注意。

Perl 在填写填满字段时，会由变量中“尽量”抓出合理的文字，以填满字段。这些文字将由原先的变量中移出。事实上在填写完某个字段之后，变量的内容已经被改变了。

到目前为止，这和一般的文本字段没有太大的不同，Perl 只是试着尽量把字段填满。当我们必须用多行显示某个变量的内容时，就能看到此种字段最大的好处了。下面的例子可以说明这一点：

```
format  PERSON =
Name：@<<<<<<<<<<<<<<<<<<  Comment：^<<<<<<<<<<<<<<
$name,                                  $comment
                                          ^<<<<<<<<<<<<<<
                                        $comment
                                          ^<<<<<<<<<<<<<<
```

```
                                $comment
                                  ^<<<<<<<<<<<<<<
                                $comment
.
```

注意，$comment 变量在此一共出现了四次。第一行格式定义会打印出人名和$conment 变量内容的最前面几个字，就在此时这几个字已从$comment 中“挖”走了。第二个定义也会参考到$comment 变量，同样地再显示其中前面几个字，就这样$comment 变量里的字符串愈来愈短。利用这个方法，可有效地把$comment 里面的内容平均分成四行打印出来。

你可能会问，如果全部字符串少于四行怎么办呢?答案是可能会有一或两个的空白。这虽然不是什么大问题，不过有时候还是会造成困扰。要解决这个问题，可以在每个字段行前面加上一个“~”符号，此行输出若为空白，会强制不打印出来。“~”号本身会被当成空白打印出，因此可放在任何应为空白所在的位置。这样，我们可以改写上面例子如下：

**程序清单 12-11：填满字段的使用**

```
format  PERSON =
Name: @<<<<<<<<<<<<<<<<<<< Comment: ^<<<<<<<<<<<<<<
$name,                              $comment
  ~                                   ^<<<<<<<<<<<<<<
                                    $comment
  ~                                   ^<<<<<<<<<<<<<<
                                    $comment
  ~                                   ^<<<<<<<<<<<<<<
                                    $comment
.
```

现在，如果$comment 里面的字符串只有第二行，第三和第四行空白会自动忽略。

如果字符串超过四行呢？消极的方法是用上面那样的写法写满 20 行，希望这能够用。但是 Perl 提供有更强大的功能：我们只要在某个字段行最前面加上两个“~”符号，就可代表不停重复此行，直到输出结果全为空白为止。这样上面的程序就可以简化为：

**程序清单 12-12：程序清单 12-11 的简化**

```
format  PERSON =
Name: @<<<<<<<<<<<<<<<<<<< Comment: ^<<<<<<<<<<<<<<
$name,                              $comment
  ~~                                  ^<<<<<<<<<<<<<<
                                    $comment
```

注意，停止条件是某处的输出全部为空白，所以变量字符串里面不能有常数，否则不会停止。

### 12.3.5 其他问题

1. 输出到其他文件

缺省时，函数 write 将结果输出到标准输出文件 STDOUT，我们也可以通过使用参数让它将结果输出到任意其他的文件中。最简单的方法就是把文件变量作为参数传递给 write，如：

```
write (MYFILE);
```

这样，write 就用名为 MYFILE 的打印格式输出到文件 MYFILE 中。但是这样就不能用$~变量来改变所使用的打印格式了，因为系统变量$~只对缺省文件变量起作用。但我们花可以改变缺省文件变量，改变$~，再调用 write，例如：

```
select (MYFILE);
$~ = "MYFORMAT";
write;
```

当 select 改变缺省文件变量时，它返回当前缺省文件变量的内部表示，这样我们就可以创建子程序，按自己的想法输出，又不影响程序的其他部分，如：

**程序清单 12-13：用子程序输出**

```
sub write_to_stdout
{
    local ($savefile, $saveformat);
    $savefile = select(STDOUT);
    $saveformat = $~;
    $~ = "MYFORMAT";
    write;
    $~ = $saveformat;
    select($savefile);
}
```

2. 分页

在输出到打印机时，可以在每页顶部输出相应的信息，这样的特殊文本叫页眉。定义页

眉实际上就是定义名为 filename_TOP 的打印格式，例如，给标准输出文件定义页眉如下：

```
format STDOUT_TOP =
    Consolidated Widgets Inc. 1994 Annual Report.
```

在页眉的定义中也可以包含值域，页眉中经常使用的一个特殊值是当前页码，存储在系统变量$%中，如：

```
format STDOUT_TOP =
Page @<<.
$%
```

我们也可以通过改变系统变量$^去改变定义页眉的打印格式名，与$~一样，$^只对当前缺省文件起作用，因此可以与 select 函数结合使用。

缺省情况下，每页长度为 60 行，可以通过改变$=来改变页长，如：

```
$= = 66;     #页长设为66行
```

此赋值语句必须出现在第一个 write 语句前。

注意，一般使用分页机制时不用 print 函数，因为当用 write 输出时，Perl 解释器跟踪每页的当前行号。如果必须使用 print 而又不打乱页计数，可以调整系统变量$-。$-的含义是当前行到页末之间的行数，当$-达到零时，就开始新的一页，调整方法如：

```
print ("Here is a line of output\n");
$- -= 1;
```

## 12.4　小结与练习

### 12.4.1　小结

本章介绍了输入输出处理。输入输出操作是编程语言中最常用的操作，几乎每一个程序都离不开输出输出操作。Perl 5 是输入输出操作最丰富的语言之一。

本章第 1 节介绍了基本输入。基本输入就是来自<STDIN>的输入，输入时可以使用输入操作符。输入有时还要用到@ARGV 数组，它是一个标准的 Perl 5 数组，是由 Perl 5 解释器在启动时生成的。Perl 5 自动取得命令行中的 Perl 程序名之后和按 Enter 键之前的所有项目并使用它来生成数组@ARGV。

第 2 节介绍了基本输出。除了前面已经很熟悉的 print 函数之外，还介绍了两个格式化输出函数 sprintf 和 printf，它们的作用相当于 C 语言中的 printf 函数。

第 3 节是关于另外一种不使用上面两个函数的格式化输出，它由 format 来自定义格式，比 sprintf 和 printf 函数的功能更加丰富，可以针对不同的领域定义不同的格式。其中需要用到字段容器，字段容器有四种：即文本字段、数字字段、多行字段和填满字段。

本章最后一节介绍了关于输入输出的一些其他问题，包括如何将数据输出到其他文件和在打印时如何分页。

学习了本章，读者应该掌握 Perl 中的输入输出操作，掌握了它，程序就具备了良好的输入输出接口。下一章将重点介绍 Perl 的包和模块。

### 12.4.2 习题与思考

（1）写一个程序，读入一个包含多行数据的文件，每行由用户名、公司名和 E-mail 地址组成（它们之间用逗号分隔），下面是数据示例：

John Doe:Foo Technologies:jdoe@footech.com

用格式化的形式打印出每个域，要求使用 format 和 write。

（2）在上面程序的基础上，为输出加上一个页眉（top-of-page）。（如果你的文件很短，就把每页的长度设得小一点，以便有多页）。

（3）在上两个程序的基础上，在页眉上加上一个指示页数的数字，以便可以得到 page 1, page 2 等等。

（4）写一个程序，使它完成与 type 命令类似的功能，但需要将指定文件中的各行颠倒过来显示。如果没有指定文件，就将从标准输入中读入的各行颠倒。

# 第13章

## 包和模块

使用 Perl 包
使用 Perl 模块
Perl 标准模块
小结与练习



在 UNIX 或 Linux 下有许多工具，如 awk、sed、shell 等等，这些语言都不能用来开发比较大的项目，原因之一就是这些语言缺少对模块编程的支持。当使用这些语言开发程序的时候，根本就没有现成的代码可以复用，所有的代码都必须自己去写。

相反，Perl 语言却比较成功，原因是可以使用许多现成的第三方的模块（或者称为程序库）。当评价一门语言的时候，语言本身的性能固然很重要，但是更为重要的是有多少现成的模块可供使用。

Perl 可以把代码分成多个可复用的模块。本章将介绍 Perl 语言的包和模块。

本章要点：

- 使用 Perl 包。
- 使用 Perl 模块。
- Perl 标准模块介绍。

# 13.1 使用 Perl 包

## 13.1.1 包的定义

Perl 程序把变量和子程序的名称存储在符号表中，Perl 符号表中名字的集合就称为包（package）。

package 关键字指示了一个新的名字空间的开始。从这以后定义的所有变量和子程序的名字都存储在该包关联的符号表中，直到遇到另一个 package 语句为止。下面是一个例子：

```
package BankAccount;
$total = 0;
sub deposit {
    my ($amount)= @_;
    $total += $amount;
    print "You now have $total dollars \n";
}
sub withdraw {
    my ($amount)= @_;
    $total -= $amount;
    $total = 0   if $total < 0;
    print "You now have $total dollars \n";
}
```



用户定义的$total 变量、deposit 、withdraw 函数等都属于 BankAccount 包。



读者可能在某些源程序中看不到 package 关键字，那是因为，在没有显式的 package 声明语句的情况下，Perl 编译器假定这个包的名字是 main。

在一个程序中可以定义多个包，每个包有一个单独的符号表。

每个符号表有其自己的一组变量、子程序名，各组名字是不相关的，因此可以在不同的包中使用相同的变量名，而代表的却是不同的变量。

如：

```
$var = 14;
package mypack;
$var = 6;
```

第一个语句创建变量$var 并存储在 main 符号表中，第三个语句创建另一个同名变量$var 但存储在 mypack 包的符号表中。



存储变量和子程序名字的缺省符号表是与名为 main 的包相关联的。如果在程序里定义了其他的包，当需要切换回去使用缺省的符号表时，可以重新指定 main 包，如下所示：

```
package  main;
```

这样，接下来的程序就好像从没定义过包一样，变量和子程序的名字像通常那样存储。

### 13.1.2　包的引用和导入

在一个包中可以引用其他包中的变量或子程序，方法是在变量名前面加上包名和一个单引号，如下所示：

```
package mypack;
$var = 26;
package main;
print ("$mypack'var\n");
```

这里，$mypack'var 为 mypack 包中的变量$var。



在 Perl 5 中，包名和变量名用双冒号隔开，即$mypack::var。

有些时候，为了输入更少的代码可能需要把某个符号导入到你的名字空间。比如，为了使用 BankAccount 包中的 deposit 函数，必须使用如下的语句：

```
BankAccount::deposit;
```

但是为了输入上的方便，读者可能想仅使用“deposit”来调用这个函数，这时候可以使用 use 语句来指定需要导入的函数列表。

如下所示：

```
use BankAccount ('withdraw', 'deposit');
withdraw();  # Can now call function without fully qualifying it.
```

### 13.1.3 包之间的切换

在程序里可以随时在包间进行切换，如：

```
#! /usr/bin/Perl
package pack1;
$var = 26;
package pack2;
$var = 34;
package pack1;
print ("$var\n");
```

运行结果如下：

```
$program
26
$
```

第二行定义了包 pack1，第三行创建变量$var，存储在包 pack1 的符号表中；第四行定义新包 pack2，第五行创建另一个变量$var，存储在包 pack2 的符号表中。这样就有两个独立的$var，分别存储在不同的包中。第六行又指定 pack1 为当前包，因为包 pack1 已经定义，这样，所有变量和子程序的定义和调用都为该包的符号表中存储的名字。因此第七行对$var 的调用为 pack1 包中的$var，其值为 26。



在 Perl 5 中，可以用如下语句指定无当前包：

```
package;
```

这时，所有的变量必须明确指出所属包名，否则为无效错误。

```
$mypack::var = 21;    #ok
    $var = 21;    #error - no current packages
```

这种情况直到用 package 语句指定当前包为止。

### 13.1.4　包和子程序

包的定义将影响到程序中的所有语句（当然也包括子程序），请看下面的例子：

```
  package mypack;
subroutine mysub {
      local ($myvar);
   # stuff goes here
}
```

这里，mysub 和 myvar 都是包 mypack 的一部分。在包 mypack 外调用子程序 mysub，则需要指定包名，如$mypack`mysub'或者是 mypack::mysub。

同时还可以在子程序中切换包，如下所示：

```
package pack1;
subroutine mysub {
   $var1 = 1;
   package pack2;
   $var1 = 2;
}
```

这段代码创建了两个变量$var1，一个在包 pack1 中，一个在包 pack2 中，包中的局域变量只能在其定义的子程序等语句块中使用，像普通的局域变量一样。

### 13.1.5　用包定义私有数据

包通常是用在含有子程序和子程序所使用的全局变量所在文件中，为子程序定义这样的包，可以保证子程序使用的全局变量不能在其他地方使用，这样的数据即为私有数据；同时，它还可以保证包的名字不能在其他地方使用。

下面看一个关于私有数据的例子，如程序清单 13-1 所示。

**程序清单 13-1：private_data.pl**

```
1 : package privpack;
2 : $valtoprint = 46;
3 :
```

```
4 : package main;
5 : # This function is the link to the outside world.
6 : sub printval {
7 :     &privpack'printval();
8 : }
9 :
10: package privpack;
11: sub printval {
12:     print ("$valtoprint\n");
13: }
14:
15: package main;
16: 1; # return value for require
```

这个子程序只有在调用 printval 后才能产生输出。该文件分为两个部分：与外界联系的部分和私有部分。前者为缺省的 main 包，后者为 privpack 包。第 6~8 行定义的子程序 printval 可被其他程序或子程序调用。printval 函数输出变量$valtoprint 的值，此变量仅在包 privpack 中定义和使用。为确保其他程序用 require 语句包含它后能正常工作，第 15 行将当前包设置为缺省包 main，第 16 行返回非零值以便使用 require 语句时不会出错。



上面这段程序中，每一个语句前面的行号在编程时是不需要的，只是在这里为了讲解的方便而人为地添加了行号。

### 13.1.6 访问符号表

在 Perl 程序中可以获得关于符号表内容的信息。

在程序中查找符号表可用数组%_package，此处_package 为需要访问的符号表所属的包名。例如%_main 含有缺省的符号表，下面语句显示了包中的所有符号名字。

```
foreach $name (keys %main::)
{
    print "$name, \n";
}
```

下面看一个例子，如程序清单 13-2 所示。

**程序清单 13-2**：access_symtable.pl

```
#!/usr/bin/Perl
```

```
#Dumping All Symbols in a Package
package DUMPVAR;
sub dumpvar {
        my ($packageName) = @_;
        local (*alias);                 # a local typeglob
    # We want to get access to the stash corresponding to the package
    # name
        *stash = *{"${packageName}::"};  # Now %stash is the symbol table
    $, = " ";                           # Output separator for print
        # Iterate through the symbol table, which contains glob values
    # indexed by symbol names.
    while (($varName, $globValue) = each %stash) {
            print "$varName ============================= \n";
           *alias = $globValue;
           if (defined ($alias)) {
             print "\t \$$varName $alias \n";
           }
           if (defined (@alias)) {
             print "\t \@$varName @alias \n";
           }
           if (defined (%alias)) {
             print "\t \%$varName ",%alias," \n";
             }
       }
}
```

下面一小段程序演示了如何使用 DUMPVAR。

```
package XX;
$x = 10;
@y = (1,3,4);
%z = (1,2,3,4, 5, 6);
$z = 300;
DUMPVAR::dumpvar("XX");
```

最后输出的结果如下：

```
x =============================
        $x 10
```

```
y ============================
        @y 1 3 4
z ============================
        $z 300
        %z  1 2 3 4 5 6
```

## 13.2 使用 Perl 模块

一般大型的程序项目都会被分割成多个部件，每一个部件通常含有一个或多个子程序及相关的变量，用以执行特定的一个或多个任务。集合了变量和子程序的部件称为程序模块。

### 13.2.1 创建模块

在 Perl 中使用包来创建模块，方法是创建包并将它存在同名的文件中。例如，名为 mymodule 的包存储在文件 mymodule.pm 中（扩展名.pm 表示 Perl Module）。下例的模块 Mymodule 含有子程序 myfunc1、myfunc2 和变量$myvar1、$myvar2。

**程序清单 13-3：mymodulepl**

```
1 : #!/usr/bin/Perl
2 :
3 : package Mymodule;
4 : require Exporter;
5 : @ISA = qw(Exporter);
6 : @EXPORT = qw(myfunc1 myfunc2);
7 : @EXPORT_OK = qw($myvar1 $myvar2);
8 :
9 : sub myfunc1 {
10:    $myvar1 += 1;
11: }
12:
13: sub myfunc2 {ss
14:    $myvar2 += 2;
15: }
```

这个程序的第 3 到 7 行是标准的 Perl 模块定义方式。第 3 行定义包，第 4 行包含内置 Perl 模块 Exporter，6、7 行进行子程序和变量的输出以及与外界联系。第 6 行创建名为@EXPORT

的特殊数组，该数组中的子程序可以被其他程序调用，这里，myfunc1 和 myfunc2 可以被访问。其他任何在模块中定义但没有赋给数组@EXPORT 的子程序都是私有的，只能在模块内部调用。第 7 行创建另一个名为@EXPORT_OK 的特殊数组，其中含有可被外部程序访问的变量，这里含有$myvar1 和$myvar2。

### 13.2.2　导入模块

使用 use 语句可将模块导入 Perl 程序中，如下句导入了 Mymodule 模块：

```
use mymodule;
```

这样，模块 Mymodule 中的子程序和变量就可以使用了。取消导入模块使用 no 语句，如下句取消了 Mymodule 模块的导入：

```
no Mymodule;
```

下面再看一个导入模块和取消导入的例子。使用 integer 模块要求所有数字运算基于整数，浮点数在运算前均被转化为整数。

```
1: #!/usr/local/bin/Perl
2:
3: use integer;
4: $result = 2.4 + 2.4;
5: print ("$result\n");
6:
7: no integer;
8: $result = 2.4 + 2.4;
9:print ($result\n");
```

程序输出结果如下：

```
$program
4
4.8
$
```

如果 use 或 no 语句出现在语句块中，则只在该块的有效范围内起作用，如：

```
use integer;
$result1 = 2.4 + 2.4;
if ($result1 == 4)
{
no integer;
$result2 = 3.4 + 3.4;
```

```
}
$result3 = 4.4 + 4.4;
```

输出结果如下：

```
4
6.8
8
```

这里，no 语句只在 if 语句中有效，在 if 语句外面仍使用 integer 模块，因此数值 4.4 在做加法前被转化成了 4。

### 13.2.3 模块使用举例

学习编程最好的办法就是自己动手，通过实践去理解事物的原理。下面我们去实际操作一个比较大的例子。

很显然，模块的使用涉及到两个方面，一个是模块的编写，另外一个是模块的使用，所以，这里也分为两部分给出这个例子。

1. 模块的编写

这个模块的名称叫做 Finance，是用来提供对简单的借贷款的计算。通过使用 Finance 模块可以使计算变得很简单。程序清单 13-4 就是这个模块的实现代码：

**程序清单 13-4：finance.pl**

```
package Finance;
require Exporter;
@ISA = (Exporter);
=head1 Finance.pm
Financial Calculator - Financial calculations made easy with Perl
=head 2
use Finance;
$pv = 10000.0;
$rate = 12.5 / 12; # APR per month.
$time = 360 ;        # months for loan to mature
$fv = FutureValue();
print $fv;
=cut
@EXPORT = qw(FutureValue,
                    PresentValue,
```

```
                    FVofAnnuity,
                    AnnuityOfFV,
                    getLastAverage,
                    getMovingAverage,
                    SetInterest);
#
# Globals, if any
#
local $defaultInterest = 5.0;
sub Finance::SetInterest($) {
        my $rate = shift(@_);
        $defaultInterest = $rate;
        printf "\n \$defaultInterest = $rate";
}
# ---------------------------------------------------------------
# Notes:
# 1. The interest rate $r is given in a value of [0-100].
# 2. The $n given in the terms is the rate at which the interest
#        is applied.
#
# ---------------------------------------------------------------

# ---------------------------------------------------------------
# Present value of an investment given
# fv - a future value
# r  - rate per period
# n  - number of period
# ---------------------------------------------------------------
sub Finance::FutureValue($$$) {
        my ($pv,$r,$n) = @_;
        my $fv = $pv * ((1 + ($r/100)) ** $n);
        return $fv;
}
# ---------------------------------------------------------------
# Present value of an investment given
# fv - a future value
# r  - rate per period
# n  - number of period
```

```
# ---------------------------------------------------------------
sub Finance::PresentValue($$$) {
        my $pv;
        my ($fv,$r,$n) = @_;
        $pv = $fv / ((1 + ($r/100)) ** $n);
        return $pv;
}
# ---------------------------------------------------------------
# Get the future value of an annuity given
# mp - Monthly Payment of Annuity
# r  - rate per period
# n  - number of period
# ---------------------------------------------------------------
sub FVofAnnuity($$$) {
        my $fv;
        my $oneR;
        my ($mp,$r,$n) = @_;
        $oneR = ( 1 + $r) ** $n;
        $fv = $mp * ( ($oneR - 1)/ $r);
        return $fv;
}
# ---------------------------------------------------------------
# Get the annuity from the following bits of information
# r  - rate per period
# n  - number of period
# fv - Future Value
# ---------------------------------------------------------------
sub AnnuityOfFV($$$) {
        my $mp; # mp - Monthly Payment of Annuity
        my $oneR;
        my ($fv,$r,$n) = @_;
        $oneR = ( 1 + $r) ** $n;
        $mp = $fv * ( $r/ ($oneR - 1));
        return $mp;
}
# ---------------------------------------------------------------
# Get the average of the last "n" values in an array.
# ---------------------------------------------------------------
```

```
# The last $count number of elements from the array in @values
# The total number of elements in @values is in $number
#
sub getLastAverage($$@) {
        my ($count, $number, @values) =    @_;
        my $i;
        my $a = 0;
        return 0 if ($count == 0);
        for ($i       = 0; $i< $count; $i++) {
                $a += $values[$number - $i - 1];
                }
        return $a / $count;
        }
# ---------------------------------------------------------------
# Get a moving average of the values.
# ---------------------------------------------------------------
# The window size is the first parameter, the number of items in the
# passed array is next. (This can easily be calculated within the
# function using the scalar() function, but the subroutine shown here
# is also being used to illustrate how to pass pointers.) The reference to the
# array of values is passed next, followed by a reference to the place
# the return values are to be stored.
#
sub getMovingAve($$\@\@) {
        my ($count, $number, $values, $movingAve) =    @_;
        my $i;
        my $a = 0;
        my $v = 0;
        return 0 if ($count == 0);
        return -1 if ($count > $number);
        return -2 if ($count < 2);
        $$movingAve[0] = 0;
        $$movingAve[$number - 1] = 0;
        for ($i=0; $i<$count;$i++) {
                $v = $$values[$i];
                $a += $v / $count;
                $$movingAve[$i] = 0;
                }
```

```
        for ($i=$count; $i<$number;$i++) {
                $v = $$values[$i];
                $a += $v / $count;
                $v = $$values[$i - $count - 1];
                $a -= $v / $count;
                $$movingAve[$i] = $a;
                }
        return  0;
        }
1;
# end of Finance module
```

2. 模块中的函数调用

其实，使用模块，也就是使用模块所提供的函数。
下面首先看看 FutureValue 函数如何使用，如程序 13-5 所示。

**程序清单 13-5：useFinance.pl**

```
#!/usr/bin/Perl -w
push(@Inc,'pwd');
use Finance;
$loan = 5000.00;
$apr =  3.5;    #  APR
$year = 10;    #  in years.
# ------------------------------------------------------------
# Calculate the value at the end of the loan if interest
# is applied every year.
# ------------------------------------------------------------
$time = $year;
$fv1 = Finance::FutureValue($loan,$apr,$time);
print "\n If interest is applied at end of year";
print "\n The future value for a loan of \$" . $loan . "\n";
print " at an APR of ", $apr , " for ",   $time, " years";
printf " is %8.2f \n" , $fv1;
# ------------------------------------------------------------
# Calculate the value at the end of the loan if interest
# is applied every month.
# ------------------------------------------------------------
```

```
$rate = $apr / 12;    # APR
$time = $year * 12; # in months
$fv2 = Finance::FutureValue($loan,$rate,$time);
print "\n If interest is applied at end of each month";
print "\n The future value for a loan of \$" . $loan . "\n";
print " at an APR of ", $apr , " for ",   $time, " months";
printf " is %8.2f \n" , $fv2;
printf "\n The difference in value is %8.2f", $fv2 - $fv1;
printf "\n Therefore by applying interest at shorter time periods";
printf "\n we are actually getting more money in interest.\n";
```

把上面的程序命名为 testFiModule.pl ，执行这个程序，结果如下：

```
# testFiModule.pl
If interest is applied at end of year
The future value for a loan of $5000
at an APR of 3.5 for 10 years is   7052.99
If interest is applied at end of each month
The future value for a loan of $5000
at an APR of 3.5 for 120 months is   7091.72
The difference in value is      38.73
Therefore by applying interest at shorter time periods
we are actually getting more money in interest
```

## 13.3　Perl 标准模块

Perl 的成功，其中一个非常重要的原因就是它的模块化设计，也就是说，有许多现成的模块可以使用，这样可以大大的提高项目开发的效率。

在这一节里，我们将给大家介绍许多标准模块，各个模块可实现不同的功能，读者可以从中以找出对自己有用的模块。

（1）AnyDBM_File

这是一个纯虚基类，为其他数据库模块所继承，提供一个标准的 DBM 数据库接口。

（2）AutoLoader

此模块与 AutoSplit 模块以及一个_END_符号一起使用，可以推迟子程序装载的时间。主程序运行前并不加载子程序，直到子程序被调用的时候加载。使用 AutoLoader 模块，用户必须把子程序的定义放在_END_符号之后，用户还可以手动运行 AutoSplit 模块，把子程序定义

分割成分离的文件。

（3）AutoSplit

此模块把程序分解为多个文件，以便 AutoLoader 模块处理。

（4）Autouse

延后载入模块，直到被调用为止。

（5）B

Perl 的编译器和一些工具。B 模块通常用于编写实现新的编译器。

（6）B::Asmdata

自动生成 Perl 操作码，用于生成字节码。

（7）B::Assembler

汇编 Perl 的字节码。

（8）B::Bblock

此模块被 B::CC 用来分析代码的“基本代码段”。基本代码段指的是没有跳转、没有分支的代码段。

（9）B::Bytecode

Perl 编译器的字节码后端编译器。

（10）B::C

Perl 编译器的 C 语言代码后端编译器。

（11）B::CC

Perl 编译器的优化 C 语言后端编译器。此编译器根据程序流程，将 Perl 源代码转化生成 C 语言源代码。

（12）B::Debug

检查 Perl 语法数，输出关于字节码的有关调试信息。

（13）B::Deparse

反向编译，根据 Perl 中间格式代码（字节码、操作码）反向编译为 Perl 源代码。

（14）B::Disassembler

反汇编 Perl 字节码。

（15）B::Lint

此模块相当于一个扩展的 Perl –w。-w 选项激活 Perl 代码编译过程中的所有警告。

（16）B::Showlex

显示在文件、函数中使用的所有有名变量。

（17）B::Terse

遍历 Perl 语法树，并输出简洁的操作码信息。

（18）B::Xref

生成 Perl 程序中交叉引用的报告。

（19）Benchmark

提供一系列的方法来计算程序、代码的运行时间。提供了如下一些函数：

new　创建新的对象，并赋予当前时间，如$tf = new Benchmark

debug 设置 debug 标志，启用或者禁用 debug

iters 返回反复次数

timeit()　运行一个代码，计算运行时间

timethis() 反复运行一个代码

timethese() 反复运行几个代码段

timediff() 返回两个 Benchmark 之间的差别

timestr 格式化 timediff()函数返回的字符串

clearcache()　清除计算空循环的缓冲时间

clearallcache()清除所有缓冲时间

cmpthese() 输出 timethese()函数的比较结果

countit()　在给定的时间内，代码段运行的次数

disablecache()　禁止空循环计时的缓冲，这个函数强制 Benchmark 重新计算每一段代码的运行时间

enablecache() 允许空循环计时的缓冲，空循环的计时将只进行一次

timesum() 计算两个 Benchmark 的和

（20）Carp

Carp 模块的作用和 die()、warn()类似，不过它的错误报告更精确，同时还可以报告程序出现错误的位置。

（21）CGI

用于生成 Web 页、表单等，支持文件上传、Cookies、服务器端 push 和框架等。

（22）CGI::Carp

CGI 程序，可以向 HTTPD 或其他错误日志中写错误信息。

（23）CGI::Cookie

Cookie 是一个允许 Web 服务器在用户浏览器所在的计算机上保存一些永久信息。每一个 Cookie 都由一个名称（-name）和数据（-value），以及-expires、-domain、-path 和-secure 四个属性组成。

（24）CGI::Fast

这是一个 CGI 模块的子类，提供对 Open Market FastCGI 标准的支持。Open Market FastCGI 标准是把 CGI 脚本转化为可以在服务器上永远运行的过程，从而加快 CGI 的速度。

（25）CGI::Push

这是服务器端 Push 模块，用来创建动态页面（在规定时间内刷新内容）。提供给 CGI::Push 一个子函数的指针，这样，Push 模块每一次调用子函数时就可以生成一个页面。页面的内容传送到浏览器替代原有的内容。

这个技术不仅可用于生成 HTML 页面，也可以用于图像文件，从而生成动态的图像。

（26）Class::Struct

在 Perl 中以 Perl 类的形式声明类似于 C 语言中的 struct 类型的数据结构，此模块提供一个 struct()函数，其定义如下：

struct(CLASS_NAME=>[ELEMENT_LIST]);

struct(CLASS_NAME=>{ELEMENT_LIST});

struct(ELEMENT_LIST);

（27）Config

此模块获得 Perl 的设置信息，Perl 的设置信息保存在 config.sh 或者 config.pm 文件中。Config 模块提供了三个函数：

myconfig()　　返回主要的 Perl 设置信息

config_sh()　　返回全部的 Perl 设置信息

config_vars(@names)　输出给定的设置名称的值

（28）constant

在 Perl 中定义常量的方法。其常量可以为数值、标量或者是散列数值。实数常量的精确度在于运行脚本的平台，常量定义如下：

use constant PI=>3.1415926;s

（29）CPAN

CPAN（Comprehensive Perl Archive Network）是一个完全的 Perl 文档网络，提供了大量的源代码、文档、脚本第三方模块和扩展模块，内容涵盖从商业数据库接口到键盘和屏幕的控制等各个方面。使用 CPAN 提供的功能，你可以从 CPAN 站点上查询、下载 Perl 模块，并可以自动编译和安装有关的 Perl 模块和扩展功能。

（30）CPAN::FirstTime

CPAN 模块的工具，配置文件初始化信息，这个初始化程序询问几个问题，然后保存在 CPAN::Config 文件中。

（31）CPAN::Nox

此模块和 CPAN 差不多，其不同在于试图组织在模块运行期间对模块进行编译。

（32）Cwd

此模块提供了如下一些函数：

getcwd()　获取当前工作目录的路径名

abs_path()　获得给定目录的绝对路径名

fastcwd()　功能和 getcwd()类似，运行速度较快，可以调用 chdir()函数改变当前工作目录

Fast_abs_path()　功能和 abs_path()相同，但运行速度快，安全性不高

chdir()　改变当前工作目录

cwd()　功能同 getcwd()和 fastcwd()类似，运行稳定性高，安全性好

下面是一个使用 cwd()函数的例子：

```
use Cwd;
$dir = cwd;
use Cwd 'chdir';
chdir "/usr/local/www";
print $ENV{'PWD'};
```

（33）Data::Dumper

Perl 数据结构格式化字符串，以便打印调试，使用范例如下：

```
use Data::Dumper;
print Dumper($foo,$bar);
print Data::Dumper->Dump( [$foo,$bar],[qw(foo *ary)]ss );
```

（34）DB_File

Perl 5 用来存取 Berkeley DB 的模块。

（35）Devel::SelfStubber

该模块可以生成脚本所需要装载的模块，保证调用模块的正确位置。对于多重继承的模块来说，本模块可以保证装载正确位置的模块。

（36）DirHandle

此模块提供了 opendir()、closedir()、readdir()、rewinddir()等函数，使用此模块可以避免名字空间的混乱。使用范例如下：

```
use DirHandle;
$dir = new DirHandle ".";
if(defined $dir)
{
    while(defined($_=$dir->read))
    {
        sub1($_);
     }
     $dir->rewind;
     while(defined($_=$dir->read))
    {
        sub2($_);
     }
     undef $dir;
}
```

（37）DynaLoader

此模块定义了一个多平台下动态链接机制的标准通用接口，主要目的是执行自动动态装载的 Perl 模块。

（38）English

使用英语名称来表示符号变量。符号变量指的是系统定义的那些以非字母为名称的变量，如$_、$!等，此模块可以提供这些变量的别名定义。

（39）Env

引入环境变量，保存在 Perl 的标量变量或者是数组中。Perl 通常把环境变量保存在一个特定的散列表(%ENV)中，而 Env 模块允许将环境变量当作标量或者是数组变量进行存取。

（40）Errno

此模块定义并有条件地引出所有系统中定义的错误常量，这些错误常量通常保存在

error.h 头文件中。

（41）ExtUtils::Command

在 makefiles 等方面，代替了通用的 VNIX 命令。

（42）Fatal

此模块提供了一些替代函数，这些函数不需要检查函数的返回值，因为如果运行失败，将产生异常，而 eval()函数可以捕获异常。需要被替代的函数在此模块引入时列出。如果关键字:void 在模块引入的列表中出现，则在列表中位于:void 后的函数，只有在脚本没有检查返回值的情况下，才被生成异常的函数所取代，使用范例如下：

```
use Fatal qw/:void open close/;
# 因为open函数的返回被if语句检查，所以不会因为错误生成异常
if (open(F,"<</myfile")
{
    warn "your file:$!";
}
# 因为返回值没有被检查，所以如果运行中有错误则生成异常
close F;
```

（43）File::Basename

此模块把完整的文件名（包括路径和文件名）分解为单纯文件名和路径名等部分。使用 fileparse_set_fstype()函数指定操作系统的类型，如 MS-DOS、MacOS、AmigaOS 等，默认系统是 VNIX。

fileparse()函数把完整的文件名分解为路径、文件名和后缀三部分，而 basename()函数返回文件名，dirname()函数返回路径，如：

```
use File::Basename;
fileparse_set_fstype("VNIX");
# 文件名
$basename = basename($fullname,@suffixlist);
# 路径
$dirname = dirname($fullname);
# 完整的文件名分解为文件名、路径和后缀
($name,$path,$suffix) =
fileparse("/use/home/zanywoo/loveu.pl","\.pl");
```

（44）File::CheckTree

在一个树型目录中运行多个文件属性的检查。

（45）File::Compare

文件句柄或者文件内容的比较。

（46）File::Copy

复制文件或者是文件句柄。

（47）File::DosGlob

类似 DOS 的文件处理，并有一些加强的功能。

（48）File::Find

遍历树形文件目录。

（49）File::Path

创建或者删除目录。

（50）File::Spec

可移植的文件名的处理。

（51）File::Spec::Mac

专门为 MacOS 设计的 File::Spec 模块。

（52）File::Spec::OS2

OS/2 系统的文件操作。

（53）File::Spec::Vnix

VNIX 系统的文件操作。

（54）File::Spec::VMS

VMS 系统的文件操作。

（55）File::Spec::Win32

Win32 系统的文件操作。

（56）File::stat

Perl 内置函数 stat()的文件名称接口。

（57）FileCache

打开比系统所允许的数量还多的文件。

（58）FileHandle

提供文件句柄的对象方法。

（59）FindBin

定位 Perl 程序所在的安装目录。

（60）GDBM_File

GNU 版本的 DBM 数据库接口。

（61）IO

安装不同的 IO 模块。

（62）IO::File

提供文件句柄的对象方法。

（63）IO::Handle

提供输入、输出句柄的对象方法。

（64）IO::Pipe

提供管道的对象方法。

（65）IO::Seekable

提供输入、输出对象的查找基本方法。

（66）IO::Select

选择系统调用的面向对象接口。

（67）IO::Socket

Socket（套接口）通信机制的对象接口。

（68）IPC::Msg

VNIX Sys V Msg IPC 类。

（69）IPC::Open2

打开一个读写过程。

（70）IPC::Open3

打开一个读写以及具有错误处理的 IPC 过程。

（71）IPC::Semaphore

Sys V IPC 对象类。

IPC::SysV

Sys V IPC 常量。

（72）Math::BigFloat

任意长度的浮点运算包，具有加、减、乘、除、绝对值、比较等运算。

（73）Math::BigInt

任意长度的整数运算包，和浮点运算相比，多了取模、最大公约数、位移、位运算等运算。

（74）Math::Complex

复数运算包。

（75）Math::Trig

三角运算包，包括 tan、csc、cos、sec、cot、cotan、asin、acos、atan、sinh 等函数。

（76）NDBM_File

Berkekey VNIX 版本的 DBM 数据库接口。

（77）Net::hostent

Perl 内置函数 gethost*()的对象方法。

（78）Net::netent

Perl 内置函数 gethnet*()的对象方法。

（79）Net::Ping

提供了用于实现 ping 工具的方法。ping 用于检查远程主机是否在线。使用范例如下：

```
use Net::Ping;
$p = Net::Ping->new();
print "$rehost is alive.\n" if $p->ping($rehost);
$p->close();
```

（80）Net::protoent

Perl 内置函数 getproto*()的对象方法。

（81）Net::servent

Perl 内置函数 getserv*()的对象方法。

（82）O

通用 Perl 编译后端接口。

（83）ODBM_File

Perl 标准数据库包。

（84）Opcode

Perl 代码在执行之前先被编译成一种内部格式，如果没有错误，就开始执行。内部格式基于大量的操作码。通常这些操作码都是有效的，任何代码都能被编译。此模块允许用户定义一个操作掩码，同样也可以使命名的操作码失效。

（85）Pod::Html

转换 POD 文件为 HTML 文件。

（86）Pod::Text

转换 POD 文件为格式化的 ASCII 文件。

（87）POSIX

IEEE Std 1003.1 的 Perl 接口。

（88）Safe

在受限空间内编译并执行代码，每一个受限的空间都有各自的名字空间、操作码。

（89）SDBM_File

Perl 标准数据库包。

（90）SelectSaver

保存和恢复选定的文件句柄。

（91）SelfLoader

指的是在需要时装载函数。

（92）Shell

在 Perl 中显式运行外部命令，如：

use Shell qw(echo cat ps cp);

如果用户设置该模块的$Shell::capture_stderr 变量位 1，则该模块还会试图捕获命令运行的 STDERR 输出。

（93）Socket

此模块是 C 语言头文件 socket.h 的 Perl 版本，装载该头文件中的定义和结构。

（94）Symbol

使用 Perl 符号以及名称。

Symbol::gensym 创建一个匿名的块，并返回一个对该块的引用。该引用作为文件或者目录的句柄。可以使用 Symbol::ungensym 和以前不支持匿名块的版本兼容。

Symbol::qualify 转换不合格的符号名称为合格的变量名，如：

Symbol::qualify_to_ref 和 Symbol::qualify 类似，不过该函数返回的是一个块的引用。

Symbol::delete_package 可以用来清除整个包的名称域。

（95）Sys::Hostname

使用任何可能的方法获取主机名，如：

use Sys::Hostname;

$host = hostname;

（96）Sys::Syslog

VNIX 系统调用 syslog(3)的 Perl 接口。

（97）Text::Wrap

文本折行以便形成简单的段落。

（98）Thread::Queue

队列是一种很像列表的、保证线程安全运行的数据结构。任何线程都可以在列表的后面安全的添加元素，或者在列表的前端删除元素。

（99）Thread::Semaphore

此模块提供了一种机制用来控制对资源的访问。

（100）Thread::Signal

启动一个具有可靠信号处理的线程。使用此模块启动线程对系统的安全运行很有必要。

（101）Tie::Array， Tie::StdArray

绑定数组的基类，把数组、句柄、散列表甚至纯变量和类进行绑定，可以扩展这些变量的功能。

（102）Tie::Handle

绑定句柄的基类定义。

（103）Tie::Hash

绑定散列数组的基类。

（104）Tie::RefHash

使用本模块可以把引用作为散列表的键。

（105）Tie::Scalar， Tie::StdScalar

绑定纯变量的基类。使用此模块可以使纯变量具有更多的性能，如动态分配等。

（106）Tie::SubstrHash

提供了表的大小固定、键的长度固定、值的长度固定的散列表。

（107）Time::gmtime

此模块默认的出口重载 Perl 的内建函数 gmtime()，取而代之以返回 Time::tm 对象，并可按照 Time::tm 模块 9 个数字的名称来返回，例如$tm_obj->year();

（108）Time::Local

提供了 timelocal 和 timegm 函数。它们的作用和 localtime()、gmtime()函数正好相反，timelocal 和 timegm 接收 6 个数字（秒、分、小时、日、月、年），然后返回给定时间和与 1970 年 1 月 1 日凌晨 0 点相差的秒数。

（109）Time::localtime

此模块默认的出口重载 Perl 的内建函数 localtime()，取而代之以返回 Time::tm 对象，并可按照 Time::tm 模块 9 个数字的名称来返回，例如$tm_obj->year();

（110）Time::tm

供 Time:gmtime 和 Time::localtime 使用的内部模块，类似于 C 语言中的结构。tm 定义了

时间结构，包含 sec、min、hour、mday、mon、year、wday、yday、isdst 等 9 个数字。

（111）UNIVERSAL

所有类的基类。

（112）User::grent

此模块的默认出口重载 Perl 的内建函数 getgrent()、getgruid()、getgrnam()，返回值为 User::grent 对象。对象中包括 name、passwd、gid、members 等有名变量，可用如下的函数来获得变量的值，如：$group_obj->members()。

（113）User::pwent

此模块的默认出口重载 Perl 的内建函数 getpwent()、getpwuid()、getpwnam()，返回值为 User::pwent 对象。对象中包括 name、passwd、uid、gid、change、age、quota、commnet、class、gecos、dir、shell、expire 等有名变量，可以用下面的函数来获取变量的值，如：$pw_obj->members()。

## 13.4　小结与练习

### 13.4.1　小结

本章介绍了使用 Perl 包和模块的一些基本知识，然后在这个基础上设计了一个 Finance 模块，同时讲解了如何使用这个模块。在本章最后给出了 Perl 标准模块列表。由于 Perl 最大的成功之处在于它有许多的第三方的模块可供使用，所以本章列出的标准模块为读者提供了一个很好的索引，以便在自己的项目中使用现成的模块，加快项目开发进度。

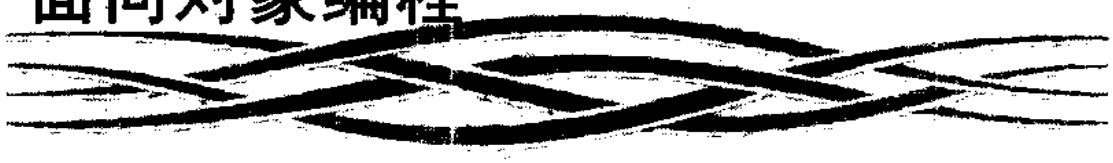
### 13.4.2　习题与思考

（1）请解释 BEGIN 和 END 关键字在模块设计中的使用。

（2）请解释下列术语：package、函数库、模块、Pragma。

# 第14章

# 面向对象编程



对象与类简介
用 Perl 5 创建类
继承
方法的重载
关于 Perl 面向对象编程的一些其他问题
小结与练习



本章将主要介绍 Perl 5 面向对象编程技术。因为本书并不是专门介绍面向对象编程的，所以这里不会就面向对象编程的每一个方面作详细的讨论，只会对相关的主题做一些探讨。Perl 一直是一个面向对象的语言，在 Perl 5 中，语法略有变动，更规范化了对象的使用。已熟悉面向对象编程的人可以在此遇到许多熟悉的术语，也可以更好地领会 Perl 5 中面向对象的编程特性，而这一点对一个想真正掌握 Perl 编程的人员来说是很重要的。所以希望对面向对象的编程技术不是很熟悉的读者，参阅一下其他专门介绍面向对象的书籍。

本章要点：

- 面向对象的思想。
- 如何用 Perl 5 创建类。
- 类的继承。
- 方法的重载。

## 14.1 对象与类简介

在具体地讲解 Perl 5 面向对象的编程之前，我们先简要介绍一下面向对象的含义。

对象可以是具体存在的事物，也可以是抽象的概念，比如说一台计算机监视器、一张桌子或是一段历史。初学者往往容易混淆对象与类，其实它们是有本质区别的。

一个类描述一类事物，描述这些事物所应具有的属性；而一个对象是类的一个实例，它具有确定的属性。比如，人类是一个类，张三是人，李四是人，大家都是人类的实例，人有身长、体重、文化程度、性别、年龄、民族等属性。张三这个具体的人是人类这个类的一个对象，它的身高 180cm，体重 70kg，大学本科学历，男，21 岁，汉族。

人类只有一个，而人类的实例可以有无数个。对象可以被创建和销毁，但类是无处不在的。

### 14.1.1 对象的特征

对象是对类中数据项的引用，所有对象都有两个与编程有关的特征。第一，可以处理或利用对象，如打开 / 关闭计算机监视器。处理对象称为操作，用函数实现。操作也称为方法，它是一个面向对象的术语，是与对象相关联的函数。对象的另一个特征是状态。计算机监视器状态可能是开或关。对象的状态用实例变量来实现。实例变量捕获对象的使其成为唯一的特性。实例变量也是一个面向对象术语，是与对象有关的变量。

### 14.1.2 对象类

类是 Perl 5 对象的容器，它是一个 Perl 包，其中包含提供对象方法的类。注意，类总是一个包，但包不一定是类。再强调一下，一个 Perl 类仅仅是一个包而已。当你看到 Perl 文档

中提到“类”时，把它看作“包”就行了。Perl 5 的语法可以创建类，如果读者已熟悉 C 语言，那么大部分类的语法读者应该不再陌生了。Perl 5 与 Perl 4 不同之处是用双冒号(::)来标识基本类和继承类（子类）。面向对象的一个重要特性是继承。Perl 中的继承特性与其他面向对象语言不完全一样，它只继承方法，而必须用自己的机制来实现数据的继承。

### 14.1.3　对象方法

Perl 5 的对象方法有两种：类方法和实例方法。

类方法以包名作为参数表中第一个参数；实例方法则以对象引用为参数表中第一个参数。类方法常用作类之间的接口程序。要调用类方法时，使用下列语法：

```
className -> methodName ( parameterList ) ;
```

Perl 5 将类名作为参数表的第一个参数。类方法的第一个语句总是将参数表中第一个参数移到词法变量中，如：

```
my  $class  =  shift ;
```

对象方法的调用语法使用向右箭头记法。向右箭头记法调用类方法并将箭头左边的参数作为传递给类方法的第一个参数。

### 14.1.4　对象变量

Perl 5 对象中有两种变量：类变量和实例变量。类变量是整个包的全局变量，在程序的运行期间持续存在。类方法和实例方法都用类变量修改对象的总体状态。对象方法用类变量生成类中所有方法间的状态和标识。类变量可以在对象存在期间随时生成，但通常在构造或对象编译期间生成。在对象存在期间尽早生成类变量可以保证调用对象方法时能随时访问类变量。类变量是类的全局变量，类的所有方法都能访问。

实例对象生成对象的唯一标识，实例变量定义对象的状态和对象的唯一性。对象的每个备份都生成和关联实例变量。有些实例变量在整个程序运行期间持续存在，但只能通过用类特别标识的引用变量访问实例变量。

### 14.1.5　对象引用变量

对象引用变量用于访问单个对象的唯一特性。它是一个包含着包范围的匿名散列的引用变量，并通过 bless 函数与包链接。

对象引用变量是由 bless 函数生成的，bless 函数建立对象与类名之间的链接。bless 函数有两个参数：对象与类名。如果不提供类名，则对象链接当前包。

bless 函数的常见用法如下：

```
bless $referenceVariable, $className;
```

但这个语法实际上隐藏了 bless 函数的一些细节。比如隐含了 bless 函数处理引用变量，但 bless 函数实际上处理的是引用变量所指向的变量，即改变了引用变量但不改变对象与类名之间的链接。如果生成对象的新引用，则不必再次将引用或对象提供给类，因为对象保持该对象与类之间的链接。

另外，尽管通常在一般使用 bless 格式时不提供左边值，但 bless 函数总是返回所链接的对象的引用。bless 函数的完整语法如下：

```
lvalue = bless object , className;
```

该语法可以将下列语句：

```
my $self = {};
bless $self, $class;
```

改变成下面的形式：

```
my $self = bless {},$class;
```

bless 函数的两句语法是 Perl 5 程序中最常见的形式。

## 14.2 用 Perl 5 创建类

如果大家了解面向对象的编程技术，就一定知道类是对同一类事物的抽象，它是对一组数据和定义在这些数据上的一组操作的封装。因此，一个类通常包含两个部分：成员变量和成员函数，前者又称作属性，后者又称做方法。其中有两种特殊的方法：构造函数和析构函数。下面就结合在 Perl 5 中创建一个类的过程来对它们分别加以介绍。

下面的例子是创建一个称为 Cocoa 的简单的类，其功能是输出一个简单的 Java 应用源码的必要部分。注意，这个例子只是对 Java 的一段源码进行输出，并不需要有 Java 的知识，其目的是加深对类的概念的理解和了解创建类的方法。

首先，创建一个名为 Cocoa.pm 的包文件（扩展名 pm 是包的缺省扩展名，意思是 Perl Module）。在做其他事之前，先加入“1;”这样一行，当你增加其他行时，记住保留“1;”为最后一行。这是 Perl 包的必需条件，否则该包就不会被 Perl 处理。下面是该文件的基本结构。

```
package Cocoa;
# Put "require" statements in for all required,imported packages
# Just add code here
1; # terminate the package with the required 1;
```

接下来，再往包里添加方法使之成为一个类。第一个需要添加的方法是 new()，它是创建对象时必须被调用的，new()方法是对象的构造函数。

### 14.2.1　构造函数

构造函数是对象初始化的方法。习惯上， Perl 5 构造器又被称为 new：

```
className -> new();
```

其他类用对象引用访问实例方法，实例方法用对象引用访问类实例变量。由于构造函数最终生成对象引用，后面供实例方法使用，因此构造函数总是类的一个方法。

1. 定义构造函数

构造函数是类的一个特殊的方法，它返回与类名相关的一个引用。将类名与引用相结合称为“祝福”一个对象，因为建立该结合的函数名为 bless()，其语法为：

```
bless YeReference [,classname]
```

YeReference 是对被“祝福”的对象的引用，classname 是可选项，指定对象获取方法的包名，其缺省值为当前包名。

一个典型的构造函数如下所示：

**程序清单 14-1：一个典型的构造函数**

```
sub new {
    my $class = shift;
    my $self  = {};          # allocate new hash for object
    bless($self, $class);
    return $self;
}
```

我们可以这样调用这个构造函数：

```
$object = Class->new();
```

构造函数返回一个与该类结合的内部结构的引用，如：

**程序清单 14-2：构造函数返回引用**

```
sub new
{
    my $this = {};      # Create an anonymous hash, and #self points to it.
    bless $this;       # Connect the hash to the package Cocoa.
    return $this;      # Return the reference to the hash.
}
1;
```

{}创建一个对不含键/值对的关联数组的引用，返回值被赋给局域变量$this。函数 bless() 取出该引用，告诉对象它引用的是 Cocoa，最后返回该引用。函数的返回值现在指向这个匿名散列表。

从 new()函数返回后，$this 引用被销毁，但调用函数保存了对该散列表的引用，因此该散列表的引用数不会为零，从而使 Perl 在内存中保存该散列表。

可以在构造函数中加入更多的代码，以完成更多的操作，但通常都是初始化操作。如在 Cocoa.pm 中，可以在每个对象创建时输出一个简单声明，还可以用构造函数初始化变量或设置数组或指针。加上声明的 Cocoa 构造函数如下：

**程序清单 14-3：在构造函数中初始化**

```
sub new {
    my $this = {};
    print "\n /* \n ** Created by Cocoa.pm \n ** Use at own risk";
    print "\n ** Did this code even get pass the javac compiler? ";
    print "\n **/ \n";
    bless $this;
    return $this;
}
```

也可以简单地调用包内或包外的其他函数来做更多的初始化工作，如：

**程序清单 14-4：在构造函数中调用其他函数进行初始化**

```
sub new {
   my $this = {}
    bless $this;
    $this->doInitialization();
    return $this;
}
```

创建类时，应该允许它可被继承，可以把类名作为第一个参数来调用 new 函数，那么 new 函数就像下面的语句：

**程序清单 14-5：在构造函数中调用其他类**

```
sub new {
    my $class = shift;      # Get the request class name
    my $this = {};
    bless $this, $class;      # Use class name to bless() reference
    $this->doInitialization();
    return $this;
}
```

2. 使用构造函数

创建对象时可调用构造函数如下：

```
$cup = new Cocoa;
```

下面语句就是使用该包创建对象的例子：

```
push (@INC,'pwd');
use Cocoa;
$cup = new Cocoa;
```

第一行中，将当前目录加到路径寻找列表@INC 中供寻找包时使用，也可以在不同的目录中创建模块并指出该绝对路径。例如，如果在/home/test/scripts/创建包，第一行语句应该如下：

```
push (@INC , "/home/test/scripts");
```

在第二行中，包含包 Cocoa.pm 以获取脚本中所需的功能。use 语句告诉 Perl 在@INC 路径寻找文件 Cocoa.pm 并包含到解析的源文件拷贝中。use 语句是使用类必须的，相当于 C 语言中的“#include”语句。

第三行调用 new 函数创建对象，这是 Perl 的妙处，也是其易混淆之处。创建对象的方法有多种，还可以这样写：

```
$cup = cocoa->new();
```

```
$cup = Cocoa::new();
```



用户可以使用下列三种方式之一来调用构造函数：

（1）new Cocoa
（2）Cocoa::new()
（3）Cocoa->new()

可以多次 bless 一个引用对象。但是，一个新的将被 bless 的类会将对象已被 bless 的引用去掉，对 C 语言和 Pascal 程序员来说，这就像把一个指针赋给分配的一块内存，再把同一指针赋给另一块内存而不释放掉前一块内存一样。总之，一个 Perl 对象每一时刻只能属于一个类。

对象和引用的真正区别是什么呢？Perl 对象被 bless 以属于某类，引用则不然，如果引用被 bless，它将属于一个类，也便成了对象。对象知道自己属于哪个类，引用则不属于任何类。

3. 实例变量

作为构造函数的 new()函数的参数叫做实例变量。实例变量在创建对象的每个实例时用于初始化，例如可以用 new()函数为对象的每个实例起个名字。

可以用匿名散列表或匿名数组来保存实例变量。

用散列表的代码如下：

**程序清单 14-6：用匿名散列表保存实例变量**

```
sub new {
    my $type = shift;
    my %parm = @_;
    my $this = {};
    $this->{'Name'} = $parm{'Name'};
    $this->{'x'} = $parm{'x'};
    $this->{'y'} = $parm{'y'};
    bless $this, $type;
}
```

用数组保存的代码如下：

**程序清单 14-7：用数组保存实例变量**

```
sub new {
```

```
    my $type = shift;
    my %parm = @_;
    my $this = [];
    $this->[0] = $parm{'Name'};
    $this->[1] = $parm{'x'};
    $this->[2] = $parm{'y'};
    bless $this, $type;
}
```

构造对象时，可以传递参数如下：

```
$mug = Cocoa::new( 'Name' => 'top','x' => 10,'y' => 20 );
```

操作符=>与逗号操作符功能相同，但=>可读性更好。访问方法如下：

```
print "Name=$mug->{'Name'}\n";
print "x=$mug->{'x'}\n";
print "y=$mug->{'y'}\n";
```



在使用构造函数时一定要注意以下几点：

（1）一定要在构造函数中初始化变量。
（2）一定要用 my 函数在构造函数中创建变量。
（3）最好不要在构造函数中使用 local，除非真的想把变量传递给其他子程序。
（4）一定不要在类模块中使用全局变量。

### 14.2.2　方法

1. 方法的定义

Perl 类的方法就是一个 Perl 函数，即通常所说的成员函数。Perl 的方法定义不提供任何特殊语法，但规定方法的第一个参数为对象或其被引用的包。Perl 中包含两种方法：静态方法和虚方法。

静态方法第一个参数为类名，虚方法第一个参数为对象的引用。方法处理第一个参数的方式决定了它是静态的还是虚的。静态方法一般忽略掉第一个参数，因为它们已经知道自己在哪个类了，构造函数就是一个静态方法。虚方法通常首先把第一个参数 shift 到变量 self 或 this 中，然后将该值作普通的引用使用。如：

下面是一个类中定义的几个方法。第一个方法是打印类中所有的项（该类中的项是以散列的形式出现的）；第二个方法是根据所给的键，得到对应的值；第三个方法设定名称。

**程序清单 14-8：在类中定义方法**

```
sub nameLister {
        my $this = shift;
        my ($keys ,$value );
        while (($key, $value) = each (%$this)) {
             print "\t$key is $value.\n";
        }
}

sub get_name {
     my $self = shift;
     return $self->{NAME};
}

sub set_name {
     my $self = shift;
     $self->{NAME} = shift;
}
```

后两个方法也可以仅仅定义一个方法，实现两个函数的功能，根据是否传入了参数来决定执行哪个动作，如：

**程序清单 14-9：将上面的两个方法合并为一个方法**

```
sub name {
     my $self = shift;
     if (@_) {
          $self->{NAME} = shift;
       }
     return $self->{NAME};
}
```

2. 方法的输出

如果你现在想引用 Cocoa.pm 包，将会得到编译错误提示未找到方法，这是因为 Cocoa.pm 的方法还没有输出。输出方法需要 Exporter 模块，在包的开始部分加上下列两行：

```
require Exporter;
@ISA = qw (Exporter);
```

这两行包含了 Exporter.pm 模块，并把 Exporter 类名加入到@ISA 数组以供查找。接下来把你自己的类方法列在@EXPORT 数组中就可以了。例如想输出方法 closeMain 和 declareMain，语句如下：

```
@EXPORT = qw (declareMain , closeMain);
```

Perl 类的继承是通过@ISA 数组实现的。@ISA 数组不需要在任何包中定义，然而，一旦它被定义，Perl 就把它看作目录名的特殊数组。它与@INC 数组类似，@INC 是包含文件的寻找路径。@ISA 数组含有类(包)名，当一个方法在当前包中未找到时，就可以到@ISA 中的包去寻找。@ISA 中还含有当前类继承的基类名。

类中调用的所有方法必须属于同一个类或@ISA 数组定义的基类。如果一个方法在@ISA 数组中未找到，Perl 就到 AUTOLOAD()子程序中寻找，这个可选的子程序在当前包中用 sub 定义。若使用 AUTOLOAD 子程序，必须用“use Autoload; ”语句调用 autoload.pm 包。AUTOLOAD 子程序尝试从已安装的 Perl 库中装载调用的方法。如果 AUTOLOAD 也失败了，Perl 再到 UNIVERSAL 类做最后一次尝试，如果仍失败，Perl 就生成关于无法解析函数的错误提示。

3. 方法的调用

调用一个对象的方法有两种方法，一是通过该对象的引用（虚方法），一是直接使用类名（静态方法），但该方法必须已被输出。现在给 Cocoa 类增加一些方法，代码如下：

**程序清单 14-10：在 Cocoa 类中增加方法**

```
package Cocoa;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(setImports, declareMain, closeMain);
# This routine creates the references for imports in Java functions
sub setImports {
    my $class = shift @_;
    my @names = @_;
    foreach (@names) {
        print "import " . $_ . ";\n";
  }
}

# This routine declares the main function in a Java script
sub declareMain {
```

```
    my $class = shift @_;
    my ( $name, $extends, $implements) = @_;
    print "\n public class $name";
    if ($extends) {
        print " extends " . $extends;
    }
    if ($implements) {
        print " implements " . $implements;
    }
    print " { \n";
}

# This routine declares the main function in a Java script
sub closeMain {
    print "} \n";
}

# This subroutine creates the header for the file.
sub new {
    my $this = {};
    print "\n /* \n ** Created by Cocoa.pm \n ** Use at own risk \n */ \n";
    bless $this;
    return $this;
}
1;
```

现在，我们写一个简单的 Perl 脚本来使用该类的方法。下面，先创建一个 Java applet 源代码骨架的脚本代码：

**程序清单 14-11：用类 Cocoa 创建一个 Java applet 源代码骨架的脚本代码**

```
use Cocoa;
$cup = new Cocoa;
$cup->setImports( 'java.io.InputStream', 'java.net.*');
$cup->declareMain( "Msg" , "java.applet.Applet", "Runnable");
$cup->closeMain();
```

这段脚本创建了一个叫做 Msg 的 Java applet，它扩展(extend)了 java.applet.Applet 小应用程序并使之可运行(runnable)，其中最后三行也可以改写如下：

```
Cocoa::setImports($cup, 'java.io.InputStream', 'java.net.*');
Cocoa::declareMain($cup, "Msg" , "java.applet.Applet", "Runnable");
Cocoa::closeMain($cup);
```

其运行结果如下：

```
/*
** Created by Cocoa.pm
** Use at own risk
*/
import java.io.InputStream;
import java.net.*;
public class Msg extends java.applet.Applet implements Runnable
{
}
```

注意：如果用->操作符调用方法（也叫间接调用），参数必须用括号括起来，如：

```
$cup->setImports( 'java.io.InputStream', 'java.net.*');
```

用双冒号调用语如下：

```
Cocoa::setImports($cup, 'java.io.InputStream', 'java.net.*');
```

也可去掉括号写成：Cocoa::setImports $cup, 'java.io.InputStream', 'java.net.*' ;

### 14.2.3　Perl 类中的成员变量

Perl 类中的成员变量也就是类中的方法要操作的数据。每一个类的对象在初始化时都会赋予这些变量特定的值以区别于其他的对象。

可以用单引号（'）操作符来引用类中的变量，其形式如：

```
$class'$member
```

在 Perl5 中，也可用双冒号（::）替代单引号来获得引用，如：

```
$class::$member
```

其结果与$class'$member 相同。

### 14.2.4 析构函数

读者已经知道，生成 Perl 5 对象的标准方法是通过 new 函数，在构造函数中完成。如果大家对其他面向对象语言比较熟悉的话，应该知道还有一种用来释放内存空间和删除对象的析构器，它还能够在程序退出之前进行必要的整理工作。在 Perl 5 中，对象析构器用函数 DESTROY 表示。

析构器并不显式地调用，而是对象在内存释放和删除之前由 Perl 5 解释器调用执行。Perl 跟踪对象的链接数目，当某对象的最后一个应用释放到内存池时，该对象就自动销毁。对象的析构发生在代码停止后，脚本将要结束时。对于全局变量而言，析构发生在最后一行代码运行之后。

如果想在对象被释放之前获取控制权，可以自定义 DESTROY()方法。DESTROY()在对象将释放前被调用，使你可以做一些清理工作。DESTROY()函数不自动调用其他 DESTROY()函数，Perl 不做内置的析构工作。如果构造函数从基类多次 bless，DESTROY()可能需要调用其他类的 DESTROY()函数。当一个对象被释放时，其内含的所有对象引用自动释放、销毁。

一般来说，不需要定义 DESTROY()函数，系统会给出一个缺省的析构函数；如果读者需要自己定义它（比如有一些特殊的处理操作），其形式如下：

```
sub DESTROY {
    # Add code here.
}
```

下面是一个析构函数的例子，该函数输出对象析构的时间：

**程序清单 14-12：用析构函数输出对象析构的时间**

```
sub DESTROY {
    my $self = shift;
    printf("$self dying at %s\n", scalar localtime);
}
```

因为多种目的，Perl 使用了简单的、基于引用的垃圾回收系统。任何对象的引用数目必须大于零，否则该对象的内存将被释放。当程序退出时，Perl 彻底地查找并销毁函数，进行垃圾回收，进程中的一切被简单地删除。在 VNIX 类的系统中，这好像是多余的，但在内嵌式系统或多线程环境中却是非常必要的。

## 14.3　继承

为了实现代码的重用，避免一些重复性的工作，可以从已经存在的类中派生出新的类，这种机制叫做继承。被继承的类也被称为父类、基类或超类；继承了别的类的类称作子类或派生类。继承是面向对象的重要特征之一，它不仅符合人们认识世界的规律，也使解决问题的方法变得更加简单了。比如人是一个类，男人和女人是它的子类。有了人这个父类，在定义男人和女人时，就不需要再重复定义人类的很多共同属性和方法——从继承的父类里可以得到，而只需添加上自己特有的属性和方法即可。继承是面向对象思想的一个重要的体现，我们在系统设计时，一定要按照从抽象到具体的关系，来构筑类的层次结构，以使问题更加简化和清晰。

定义了父类之后，子类就可以通过@ISA 数组来继承父类的方法，成员变量的继承必须明确设定。下面几个程序清单创建了两个类 Bean.pm 和 Coffee.pm，其中 Coffee.pm 类继承了 Bean.pm 的一些功能。程序演示了如何从父类继承实例变量，其方法为调用父类的构造函数并把自己的实例变量加到新对象中。

Bean.pm 代码如下：

**程序清单 14-13：被继承的父类**

```
package Bean;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(setBeanType);

sub new {
        my $type = shift;
    my $this = {};
    $this->{'Bean'} = 'Colombian';
    bless $this, $type;
    return $this;
}

sub setBeanType     # This subroutine sets the class name
{
     my ($class, $name) = @_;
     $class->{'Bean'} = $name;
```

```
    print "Set bean to $name \n";
}
1;
```

此类中，用$this 变量设置一个匿名散列表，将'Bean'类型设为'Colombian'。方法setBeanType()用于改变'Bean'的类型，它使用$class 引用获得对对象散列表的访问。

Coffee.pm 的代码如下：

**程序清单 14-14：继承父类的子类**

```
# The Coffee.pm file to illustrate inheritance.
package Coffee;
require Exporter;
require Bean;
@ISA = qw(Exporter, Bean);
@EXPORT = qw(setImports, declareMain, closeMain);
sub setCoffeeType      # set item
{
    my ($class,$name) = @_;
    $class->{'Coffee'} = $name;
    print "Set coffee type to $name \n";
 }

 sub new   # constructor
 {
     my $type = shift;
     my $this = Bean->new();
     $this->{'Coffee'} = 'Instant';      # unless told otherwise
     bless $this, $type;
     return $this;
  }
  1;
```

第 4 行的"require Bean;"语句包含了 Bean.pm 文件和所有相关函数，方法 setCoffeeType()用于设置局域变量$class->{'Coffee'}的值。在构造函数 new()中，$this 指向 Bean.pm 返回的匿名散列表的指针，而不是在本地创建一个。

下面两个语句分别为创建不同的散列表从而与 Bean.pm 构造函数创建的散列表非继承和继承的情况：

```
my $this = {};      #非继承
my $this = $theSuperClass->new();      #继承
```

下面的代码演示如何调用继承的方法：

**程序清单 14-15：调用继承的方法**

```
#!/usr/bin/Perl
push (@INC,'pwd');
use Coffee;
$cup = new Coffee;
print "\n ---------- Initial values ---------- \n";
print "Coffee: $cup->{'Coffee'} \n";
print "Bean: $cup->{'Bean'} \n";
print "\n ---------- Change Bean Type ---------- \n";
$cup->setBeanType('Mixed');
print "Bean Type is now $cup->{'Bean'} \n";
print "\n ---------- Change Coffee Type ---------- \n";
$cup->setCoffeeType('Instant');
print "Type of coffee: $cup->{'Coffee'} \n";
```

该段代码的输出结果如下：

```
---------- Initial values ----------
Coffee: Instant
Bean: Colombian
---------- Change Bean Type ----------
Set bean to Mixed
Bean Type is now Mixed ---------- Change Coffee Type ----------
Set coffee type to Instant
Type of coffee: Instant
```

上述代码中，先输出对象创建时散列表中索引为'Bean'和'Coffee'的值，然后调用各成员函数改变值后再输出。

方法可以有多个参数，现在向 Coffee.pm 模块增加函数 makeCup()，代码如下：

**程序清单 14-16：加入 Coffee.pm 模块中的函数 makeCup()**

```
sub makeCup {
    my ($class, $cream, $sugar, $dope) = @_;
```

```
    print "\n================================== \n";
    print "Making a cup \n";
    print "Add cream \n" if ($cream);
    print "Add $sugar sugar cubes\n" if ($sugar);
    print "Making some really addictive coffee ;-} \n" if ($dope);
    print "================================== \n";
}
```

此函数可有三个参数，不同数目、值的参数产生不同的结果，测试程序如下：

**程序清单 14-17：函数 makeCup()的测试程序**

```
#!/usr/bin/Perl
push (@INC,'pwd');
use Coffee;
$cup = new Coffee;
print "\n Calling with no parameters: \n";
$cup->makeCup;      # With no parameters
print "\n Calling with one parameter: \n";
$cup->makeCup('1');      # With one parameter
print "\n Calling with two parameters: \n";
$cup->makeCup(1,'2');      # With two parameters
print "\n Calling with three parameters: \n";
$cup->makeCup('1',3,'1');     # With all three parameters
```

其结果输出如下：

```
Calling with no parameters:
==================================
Making a cup
==================================
Calling with one parameter:
==================================
Making a cup
Add cream
==================================
Calling with two parameters:
==================================
Making a cup
```

```
Add cream
Add 2 sugar cubes
==================================
Calling with three parameters:
==================================
Making a cup
Add cream
Add 3 sugar cubes
Making some really addictive coffee ;-)
==================================
```

在此例中，函数 makeCup()的参数既可为字符串也可为整数，处理结果相同，你也可以把这两种类型的数据处理区分开。在对参数的处理中，可以设置缺省的值，也可以根据实际输入参数值的个数给予不同处理。

## 14.4　方法的重载

### 14.4.1　子类方法的重载

继承的好处在于可以轻松获得父类输出的方法的功能，而有时却需要对父类的方法重载以获得更具体或不同的功能。下面在 Bean.pm 类中加入方法 printType()，代码如下：

**程序清单 14-18：在 Bean.pm 模块中加入函数 printType()**

```
sub printType  {
    my $class = shift @_;
    print "The type of Bean is $class->{'Bean'} \n";
}
```

然后更新其@EXPORT 数组来输出：

```
@EXPORT = qw ( setBeanType , printType );
```

调用函数 printType()的方法有以下三法：

**程序清单 14-19：调用函数 printType()**

```
$cup->Coffee::printType();
$cup->printType();
```

```
$cup->Bean::printType();
```

输出分别如下：

```
The type of Bean is Mixed
The type of Bean is Mixed
The type of Bean is Mixed
```

为什么都一样呢？因为在子类中没有定义函数 printType()，所以实际调用的均是父类中的方法，结果当然一样。如果想使子类有其自己的 printType()函数，必须在 Coffee.pm 类中加以定义：

**程序清单 14-20：定义子类中的 printType()函数**

```
# This routine prints the type of $class->{'Coffee'}
sub printType {
    my $class = shift @_;
    print "The type of Coffee is $class->{'Coffee'} \n";
}
```

然后更新其@EXPORT 数组：

```
@EXPORT = qw(setImports, declareMain, closeMain, printType);
```

现在输出结果就变成了：

```
The type of Coffee is Instant
The type of Coffee is Instant
The type of Bean is Mixed
```

这时只有当给定了 Bean::时才调用父类的方法，否则直接调用子类的方法。

那么，如果不知道父类名该如何调用父类方法呢？方法是使用伪类保留字 SUPER::。在类方法内使用语法如：

```
$this->SUPER::function(...argument list...);
```

它将从@ISA 列表中寻找。刚才的语句用 SUPER::替换 Bean::可以写为：

```
$cup->SUPER::printType();
```

其结果输出相同，为：

```
The type of Bean is Mixed
```

### 14.4.2　操作符的重载

下面的程序中的类重载了常用的几种操作符‘+ ‘、*’、‘/’、‘<=>’、‘""’和‘0+’，以适合类的特殊操作。一般操作符重载时，是取操作符原来的语义，如‘+’还是有相加的意思，只不过这里不再是两个数相加了。

包含类的程序如下：

**程序清单 14-21：操作符的重载**

```
package FixNum;
use strict;
my $PLACES = 0;
sub new {
    my $proto    = shift;
    my $class    = ref($proto) || $proto;
    my $parent   = ref($proto) && $proto;
    my $v = shift;
    my $self = {
        VALUE   => $v,
        PLACES => undef,
    };
    if ($parent && defined $parent->{PLACES}) {
        $self->{PLACES} = $parent->{PLACES};
    } elsif ($v =~ /(\.\d*)/) {
        $self->{PLACES} = length($1) - 1;
    }  else {
        $self->{PLACES} = 0;
    }
    return bless $self, $class;
}

sub places {
    my $proto = shift;
    my $self   = ref($proto) && $proto;
    my $type   = ref($proto) || $proto;
    if (@_) {
```

```
            my $places = shift;
            ($self ? $self->{PLACES} : $PLACES) = $places;
        }
        return $self ? $self->{PLACES} : $PLACES;
}

sub _max {
        $_[0] > $_[1] ? $_[0] : $_[1] ;
    }

use overload '+'    => \&add,
             '*'    => \&multiply,
             '/'    => \&divide,
             '<=>'  => \&spaceship,
             '""'   => \&as_string,
             '0+'   => \&as_number;

sub add {
    my ($this, $that, $flipped) = @_;
    my $result = $this->new( $this->{VALUE} + $that->{VALUE} );
    $result->places( _max($this->{PLACES}, $that->{PLACES} ));
    return $result;
}

sub multiply {
    my ($this, $that, $flipped) = @_;
    my $result = $this->new( $this->{VALUE} * $that->{VALUE} );
    $result->places( _max($this->{PLACES}, $that->{PLACES} ));
    return $result;
}

sub divide {
    my ($this, $that, $flipped) = @_;
    my $result = $this->new( $this->{VALUE} / $that->{VALUE} );
    $result->places( _max($this->{PLACES}, $that->{PLACES} ));
    return $result;
}
```

```
sub as_string {
    my $self = shift;
    return sprintf("STR%s: %.*f", ref($self),
        defined($self->{PLACES}) ? $self->{PLACES} : $PLACES,
            $self->{VALUE});
}

sub as_number {
    my $self = shift;
    return $self->{VALUE};
}

sub spaceship {
    my ($this, $that, $flipped) = @_;
    $this->{VALUE} <=> $that->{VALUE};
}
1;
```

调用该类的程序为：

**程序清单 14-22：调用操作符重载类的主程序**

```
use FixNum;
FixNum->places(5);
$x = FixNum->new(40);
$y = FixNum->new(12);
print "sum of $x and $y is ", $x + $y, "\n";
print "product of $x and $y is ", $x * $y, "\n";
$z = $x / $y;
printf "$z has %d places\n", $z->places;
$z->places(2) unless $z->places;
print "div of $x by $y is $z\n";
print "square of that is ", $z * $z, "\n";
```

程序执行输出如下：

```
sum of STRFixNum: 40 and STRFixNum: 12 is STRFixNum: 52
product of STRFixNum: 40 and STRFixNum: 12 is STRFixNum: 480
STRFixNum: 3 has 0 places
```

```
div of STRFixNum: 40 by STRFixNum: 12 is STRFixNum: 3.33
square of that is STRFixNum: 11.11
```

## 14.5 关于 Perl 面向对象编程的一些其他问题

面向对象编程的最大好处就是代码重用，它用数据封装来隐藏一些复杂的代码。虽然 Perl 的包和模块通过 my 函数提供数据封装功能，但是，Perl 并不保证子类一定不会直接访问父类的变量，这大大减少了数据封装的好处。



一定要通过方法来访问类变量，而不要从模块外部直接访问类变量。

当编写包时，应该保证方法所需的条件已具备或通过参数传递给它。在包内部，应保证对全局变量的访问只用通过方法传递的引用来访问。对于方法要使用的静态或全局数据，应该在基类中用 local()来定义，子类通过调用基类来获取。有时，子类可能需要改变这种数据，这时，基类可能就不知道怎样去寻找新的数据，因此，这时最好定义对该数据的引用，子类和基类都通过引用来改变该数据。

读者将看到如下方式来使用对象和类：

```
use coffee::Bean;
```

这条语句的含义是："在@INC 数组所有目录的 Coffee 子目录来寻找 Bean.pm"。如果把 Bean.pm 移到./Coffee 目录中，上面的例子将用这一 use 语句来工作。这样的好处是可以有条理地组织类的代码。再如，下面的语句：

```
use Another::Sub::Menu;
```

等同于如下的子目录树：

```
./Another/Sub/Menu.pm
```

下面是一个比较完整的具体的例子：

有一个商店销售电脑及其组件，每个组件都有产品编号、价格和折扣等属性。顾客可以购买单独的组件，也可以根据定制的组件组装一台电脑。商店已将最后的税款加到最终价格里面去了。根据这个要求编码一个程序，这个程序的运作目的就是要求出顾客购买的任何一个组件的纯利润。

读者必须考虑到一种组件可能包含另外的组件，销售税款还与销售的组件的类型有关等，最后如果要用组件组装整机还必须收取若干的组装劳务费。

最后，我们设计了如下方式编码这个程序：

**程序清单 14-23：测试类的代码**

```
$cdrom    = new CDROM    ("Toshiba 5602");
$monitor = new Monitor ("Viewsonic 15GS");
print $monitor->net_price();
$computer = new Computer($monitor, $cdrom);
print $computer->net_price();
```

下面的包 StoreItem 将对象模型变成了 Perl 的代码：

**程序清单 14-24：定义类的包 StoreItem**

```
package StoreItem;
my $_sales_tax = 8.5;   # 8.5% added to all components's post rebate price

sub new {
    my ($pkg, $name, $price, $rebate) = @_;
    bless {
        # Attributes are marked with a leading underscore, to signify that
        # they are private (just a convention)
         _name => $name, _price => $price, _rebate => $rebate
    }, $pkg;
}

# Accessor methods
sub sales_tax {
        shift; @_ ? $_sales_tax = shift : $_sales_tax
   }

sub name {
        my $obj = shift; @_ ? $obj->{_name} = shift : $obj->{_name}
   }

sub rebate {
        my $obj = shift;
        @_ ? $obj->{_rebate} = shift : $obj->{_rebate};
```

```
}

sub price {
        my $obj = shift;
        @_ ? $obj->{_price} = shift : $obj->{_price} - $obj->_rebate};
}

sub net_price {
     my $obj = shift;
     return $obj->price * (1 + $obj->sales_tax / 100);
}
```

下面是组件包 Component：

**程序清单 14-25：组件包 Component**

```
package Component;
@ISA = qw(StoreItem);
```

下面是显示器包 Monitor：

**程序清单 14-26：显示器包 Monitor**

```
package Monitor;
@ISA = qw(Component);
# Hard-code prices and rebates for now
sub new {
       $pkg = shift; $pkg->SUPER::new("Monitor", 400, 15)
   }
```

下面是光盘驱动器包 CDROM：

**程序清单 14-27：光盘驱动器包 CDROM**

```
   package CDROM;
@ISA = qw(Component);
sub new {
       $pkg = shift; $pkg->SUPER::new("CDROM", 200, 5)
   }
```

下面是计算机包 Computer：

**程序清单 14-28：计算机包 Computer**

```
package Computer;
@ISA = qw(StoreItem);
sub new {
    my $pkg = shift;
    my $obj = $pkg->SUPER::new("Computer", 0, 0);        # Dummy value for price
    $obj->{_components} = [];        # list of components
    $obj->components(@_);
    $obj;
}

# Accessors
sub components {
    my $obj = shift;
    @_ ? push (@{$obj->{_components}}, @_)
       : @{$obj->{_components}};
}

sub price {
    my $obj = shift;
    my $price = 0;
    my $component;
    foreach $component ($obj->components()) {
        $price += $component->price();
    }
    $price;
}
```

## 14.6　小结与练习

### 14.6.1　小结

面向对象设计不仅是一个编程问题，它更是一种编程思想。

本章第 1 节重点介绍了对象与类的不同含义。本章重点介绍了 Perl 面向对象地编程思想。说明类是相关的对象方法与变量的容器；方法是面向对象的术语，它是指完成一定功能的函

数。面向对象方法有两种形式：类方法和实例方法。同时较为详细地介绍了如何用类名作为参数表中的第一个参数，调用类方法。还介绍了构造器，它主要负责返回对象引用和初始化对象变量。

第 2 节详细介绍了用 Perl 5 创建类的方法。内容涉及构造函数、使用构造函数、方法的定义与调用及析构函数等。

第 3 节介绍了面向对象编程的一个重要特性——继承。继承的特性允许对象像使用自己的方法一样使用其他类的方法。Perl 通过全局@ISA 数组实现继承。要将父类加进派生类层次树中，派生类要将父类名加进@ISA 数组中。Perl 5 采用深度优先搜索机制寻找继承方法。

第 4 节介绍了方法的重载。本章的最后，给出了一个较为复杂的实例，用以说明 Perl 中面向对象编程思想的应用。

### 14.6.2 习题与思考

（1）写一个表示人的类，要求设置得到人的名称、人的年龄等信息。并写一段代码测试你所写的类。

（2）在上一题的基础上，再写一个员工的类，使它继承人类，并写一段代码对它进行测试。

# 第15章

# CGI 编程

Web 体系结构与 CGI 程序结构
HTTP 协议
CGI 规范
CGI 模块
计数器实现
留言簿实现
小结与练习

CGI（Common Gateway Interface）是指公共网关接口，它是 Web 服务器和外部应用程序之间信息交互的标准接口。

早期的 Web 主要是静态页面的调用，由 Web 服务器使用 HTTP 协议将 HTML 文档从 Web 服务器传送到用户的 Web 浏览器。HTTP 是一种以文档描述为基础的协议，以页面为传送单位的，它很适合于组织各种静态的文档类型元素，如图片、文字以及文档之间的链接。

Web 技术发展的第二阶段是生成动态的页面。随着三层 Client/Server 结构和 CGI 标准、ISAPI 扩展、动态 HTML 语言、Java/JDBC 等技术的出现，产生了可以提供用户交互的动态 Web 文档，HTML 页除了能够显示静态的信息以外，还可以作为信息管理系统中完成对数据库交互操作的人机界面。

Web 服务器一般是安装在一台功能强大的计算机上，但是，它也许因为某些原因不能充分利用现有的计算机的处理能力为远程的用户提供更多动态、有趣的内容，CGI 规范就是为了解决这样一个问题而制定的。CGI 规范在 Web 服务器和客户浏览器之间建立了标准的信息交换方式。 使用 CGI 规范，可使得信息在浏览器和服务器之间传递，外部程序在处理之后将处理结果送回到用户的浏览器中。外部程序通常指的就是 CGI 程序、CGI 脚本、CGI 应用程序或网关。用户通过初始化，使得 CGI 程序在运行的时候，能够输出动态变化的信息。

本章要点：

- Web 体系结构与 CGI 程序。
- HTTP 协议。
- CGI 规范。
- CGI 模块
- 计数器实现。
- 留言簿实现。

# 15.1 Web 体系结构与 CGI 程序结构

## 15.1.1 Web 体系结构

WWW（World Wide Web）是世界上最大的电子信息库。用户利用浏览器可以访问 Web 站点、浏览站点上的文本和图像、查询 Web 站点的数据库服务、与 WWW 上的其他用户交互。可以说，WWW 是一个全球性的、以电子方式传递信息的交换媒体。

WWW 是建立在因特网之上的计算机网络，它使用 Internet 传输数据。图 15-1 描述了 Web 的体系结构。因特网使用 TCP/IP 协议进行通信。在 TCP/IP 的基础上，WWW 定义了自己的通信协议 HTTP，即超文本传输协议。

图 15-1 描述了浏览 WWW 的操作过程。

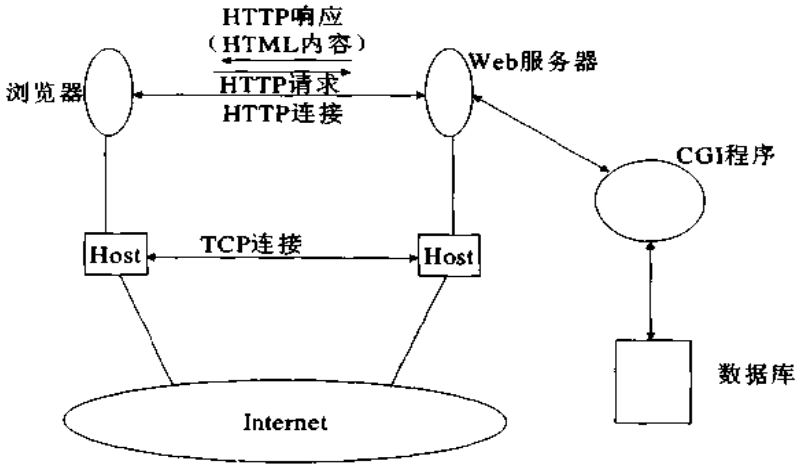



图 15-1　Web 体系结构

用户利用浏览器（如 Internet Explorer）访问 Web 站点时，首先指定该站点上的某个页面资源的地址，浏览器根据这个页面地址，产生 HTTP 请求；然后利用 Internet ，将这个 HTTP 请求发送给远程的 Web 服务器；最后 Web 服务器根据这个请求的内容，在 HTTP 响应中返回相应的页面内容。

用户利用浏览器访问的 Web 资源内容主要是 HTML 文档。HTML 文档是一种结构化的文档，它包含文档的标题，段落和表格等内容。

HTML 语言是一种标记语言，能够将一个文档的不同部分内容标记成不同结构，它利用标签 （tag） 标记文档中的不同结构内容；HTML 语言同时也是一种超文本语言，也就是说可以利用 HTML 语言创建超文本链接。

### 15.1.2　CGI 程序结构

用户浏览 WWW 时，不仅可以访问固定的 HTML 文档，还可以访问 Web 服务器上的某些程序，并在浏览器中查看这些程序的执行结果。我们把这些由用户指定，由 Web 服务器执行的程序称为 CGI 程序。

用户可以通过 HTML 表单向 CGI 程序传递参量，CGI 程序根据用户的参量，返回相应的结果。同时，利用 CGI 程序，Web 服务器还可以扩展服务内容，根据用户的需要，动态地产生页面内容。比如说，可以利用 CGI 程序访问服务器后台的数据库，根据查询数据库的结果动态产生 HTML 页面。图 15-2 描述了浏览器、HTTP 服务器和 CGI 程序三方处理一个表单的过程。

由图 15-2 可以看出，一个 CGI 程序需要完成以下三部分任务：

- HTTP 服务器接收用户输入的数据；
- 据用户输入的数据进行处理，如访问数据库；
- 将处理结构以 HTML 形式返回给 HTTP 服务器，由服务器将结果返回给用户。

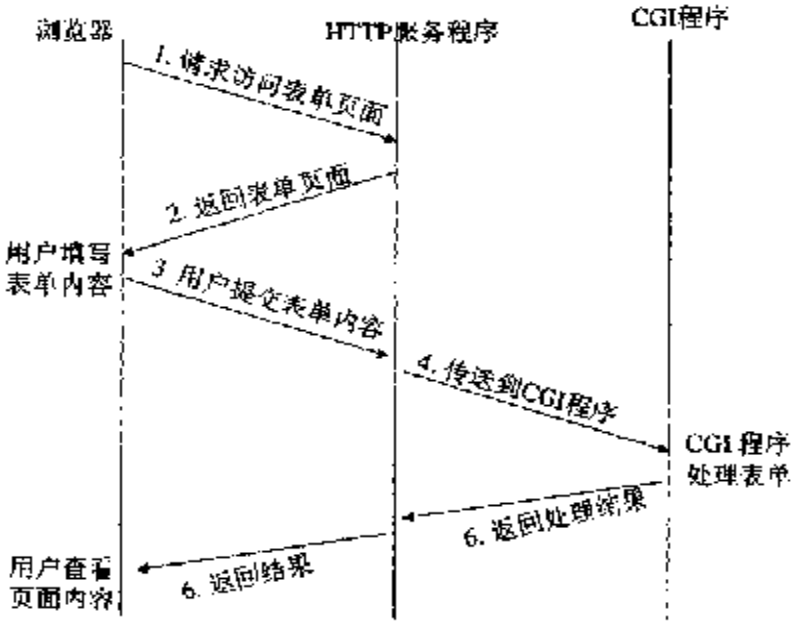



图 15-2　访问 CGI 程序的流程

CGI 本身可以实现为一个可执行程序，它可以用 C/C++、Visual Basic Script、Java Script、Voix Shell 和 Perl 等语言来实现。

下面用 Perl 来写一个最简单的 CGI 程序。程序名称为 test.pl，放在 Web 服务器的/cgi-bin/目录底下，当然也可以放在其他目录底下，只需要在 Web 服务器的配置文件中（如 Apache 服务器的 httpd.conf 文件）做相应的修改即可。

**程序清单 15-1：test.pl**

```
#!/usr/bin/Perl -w
# howdy--the easiest of CGI programs
print <<END_of_Multiline_Text;
Content-type: text/html
<HTML>
    <HEAD>
    <TITLE>Hello World</TITLE>
    </HEAD>
    <BODY>
    <H1>Greetings, Zhengwu!
    Congratulation!!
```

```
    </H1>
    </BODY>
</HTML>
END_of_Multiline_Text
```



注意，Content-type:text/html 所在行的下一行必须是空白行，这是 HTTP 协议约定的。

在浏览器的地址栏里输入 http://localhost/cgi-bin/test.pl。便可以出现如下页面：

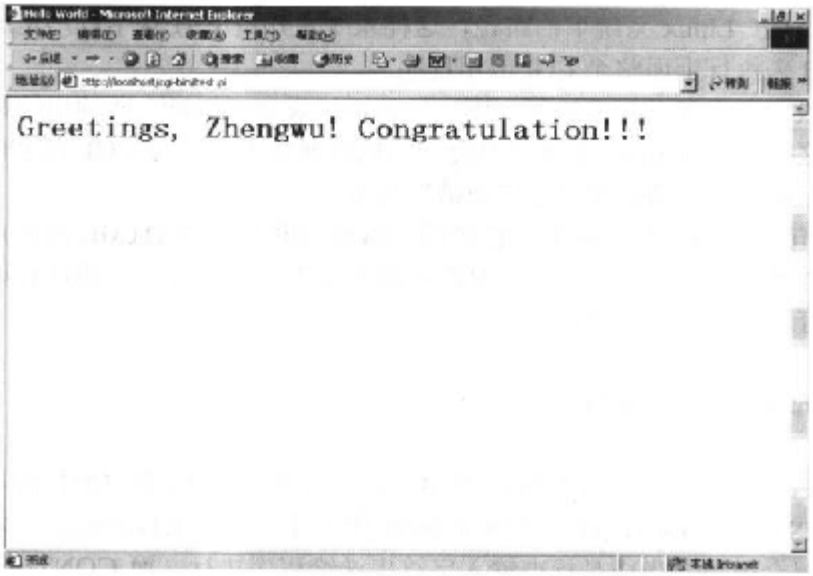



图 15-3　第一个 CGI 程序 test.pl 执行结果(Web 方式)

上面的程序显得有点简单，下面再给出一个比较经典的 Perl CGI 程序，然后详细分析其中每一行代码，这样可以使读者对 Perl CGI 编程有一个较为直观的认识。

**程序清单 15-2：demo.pl**

```
# 为了便于讲解，在每一行前面加了行号
1  #!/usr/bin/Perl
2  $|=1;
3  print "Content-type:text/html\n\n";
4  if($ENV{'REQUEST_METHOD'} eq "POST") {
5       read(STDIN,$buffer,$ENV{'CONTENT_LENGTH'});
6  }elsif ($ENV{'REQUEST_METHOD'} eq "GET") {
7      $buffer = $ENV{'QUERY_STRING'};
8  }
```

```
9   @pairs = split(/&/,$buffer);
10 foreach $pair (@pairs) {
11 ($name, $value) = split(/=/,$pair);
12 $value =~ tr/+//;
13 $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",hex($1))/eg;
14 $FORM{$name} = $value;
15 }
```

程序逐行注释如下：

1：设定 Perl 在 Linux 系统下的路径，这样系统才知道 Perl 这个程序是放在哪一个目录下，以便用 Perl 来执行上面整个 CGI 应用程序。

2：$|变量设为 1 表示关闭缓冲区。因为如果系统设定了缓冲，所输出的数据会被存在一个缓冲区中，这样，执行 print 命令时不会立刻把数据送出去。写 CGI 程序时，一般关闭缓冲区，这样可以避免顺序错乱或者是其他错误发生。

3：这条语句生成 HTTP 响应头。它告诉 CGI 应用程序要以 HTML 的格式输出，注意这里一定要有两个“\n”符号，这对应于 HTTP 响应头中的空白行。如果以普通文本格式输出的话，这一行可以用下面语句代替：

```
print "Content-type:text/plain\n\n";
```

4-5：在 Perl 语言中，用$ENV{'CGI 变量名称'}来取得 CGI 的环境变量，如$ENV{'REQUEST_METHOD'}可以得到 FORM 使用什么方式来传送数据。如果是 POST 方法，Web 服务器会把数据当成是标准输入字符串送给程序执行，而 CONTENT_LENGTH 环境变量记录了数据的长度(bytes)，如第 5 行的作用就是以标准输入(STDIN)的方式将 CONTENT_LENGTH 长度的标准输入字符串存放在$buffer 标量变量中。

6-8：如果 FORM 传送数据用的是 GET 方法，那么输入的字符串可以用 QUERY_STRING 环境变量取得，如第 7 行语句所示。

9：$buffer 变量所保存的数据是以“&”符号分隔的，为了取得一对相互对应的值，需要使用 split 函数来分隔这个字符串，如第 9 行代码所示。

10：foreach 循环将数组@pairs 里的元素一个一个指定给$pair 这个变量。

11：$pair 变量中的数据是以“=”号相连的某个数据字段的名称和其对应的值，为了取得相互对应的值，需要再次使用 split 函数进行分割。

12：因为传送数据时，会把空白的字符编码成“+”这个字符，所以要用 tr 指令把$value 中“+”这个字符解码成一个半角的空白字符。

13：在传送数据中有一些特殊字符或者是中文字符会被编码成以%位开头的连续两个十六进制值的字符串。所以这一行的功能就是用 s 取代指令把符合开头为%的连续两个十六进制数值的字符串找出来，并把它用 hex（$1 特殊变量代表符合的字符串）这个指令转换成十进制的值，最后再用 pack 函数把它还原成字符串，而本行最后的字符串 ge 是代表这个取代

函数的参数，其中 g 表示取代所有符合条件的字符串，而 e 表示要将整个取代模式的部分当成一个运算式。

14：这一行把$name 和$value 的值存在名称为 FORM 的散列数组中，以便以后使用。



这个程序所涉及到的一些知识会在以后的小节中讲到。

## 15.2 HTTP 协议

HTTP 协议，即超文本传输协议，是 WWW 使用的通信协议，Web 浏览器与 Web 服务器使用 HTTP 协议进行通信。HTTP 协议定义了浏览器和服务器之间传送数据的方式。HTTP 是处在应用层的协议，使用 MIME 处理数据，并可以适应多种媒体格式。

### 15.2.1 HTTP 通信模型

HTTP 是使用请求（request）/响应（response）模型的客户机/服务器通信模型。HTTP 客户机（浏览器）使用 URL 访问 HTTP 服务器上的某个资源。浏览器与服务器之间的通信过程可以描述如下：

- 浏览器向服务器发送一个 HTTP 请求；
- 服务器返回请求的内容（如 HTML 文档）。

浏览器在 HTTP 请求中指定要访问的资源地址和传输数据过程中所需要的信息。HTTP 服务器在 HTTP 响应中返回答复的状态和描述数据内容的信息和数据内容本身。这种请求/响应模型使用 MIME 来对所请求的数据进行编码。

HTTP 协议是无状态（stateless）的。也就是说，每一个 HTTP 请求/响应过程之间是没有联系的。浏览器与服务器每进行一次 HTTP 操作，就建立一条 TCP 连接，在完成这次操作之后，就断开 TCP 连接。一次 HTTP 操作被称为一次事务（transation），每次事务是完全独立的。

为了理解请求/响应过程，我们看一个例子，具体步骤如下：

第一步：用户在浏览器地址栏中输入 URL 地址：

http://www.somesite.edu.cn/main.html

第二步：浏览器产生一个 HTTP 请求：

GET /main.html HTTP/1.0

第三步：浏览器与主机 www.somesite.edu.cn 上的 HTTP 服务器建立连接，并发送 HTTP 请求：

第四步：服务器检查“/main.html”文件是否存在，如果不存在，服务器返回错误信息状态：HTTP/1.0 404 Not Found

Date: FRI. 06 April 2001 14:42:55 GMT

Server: Apache/2.0.1

第五步：如果文件存在，服务器确定这个文件的类型（HTML 文件、gif 图像或者是 CGI 程序）。然后服务器向浏览器返回成功状态、文件数据内容的相关信息和文件数据内容本身：

```
HTTP/1.0 404 Not Found
Date: FRI. 06 April 2001 14:42:55 GMT
Server: Apache/2.0.1
Contenet-type:  text/html
Content-length: 108
Last-modified: Thu. 02 Feb 2000 02:00:09 GMT
<html>  <head>
<title> Hello.World! </title>
</head>
<body>
<h1>Hello.World!</h1>
</body>  </html>
```

第六步：浏览器从响应的内容中提取数据内容(HTML 文档)并显示出来。

### 15.2.2 HTTP 请求

为了访问 HTTP 服务器，浏览器必须要向服务器发送 HTTP 请求。一个 HTTP 请求包括如下内容：

- 请求行。
- 请求头标。
- 一行空白行。
- 消息正文。

现在分别介绍如下：

1. 请求行

常用的请求方法有三种：

（1）GET

当使用这种方法时，CGI 程序从环境变量 QUERY_STRING 中获取数据。QUERY_STRING 把客户端的数据传递给服务器。为了解释和执行程序，CGI 要分析（处理）此字符串。当需要从服务器获得数据并且不改变服务器上的数据时，应该选用 GET。但如果字符串长度超过了一定长度，必须选用 POST 方法。



你可能会在 HTML 文件中看到 <a href = /cgi-bin/index.pl ? data> ，实际上这就是一个使用 GET 方法的例子。

（2）POST

使用 POST 方法时，WEB 服务器通过 STDIN（标准输入）向 CGI 程序传送数据。服务器在数据的最后没有使用 EOF 字符标记，因此程序为了正确的读取 stdin，必须使用 CONTENT_LENGTH 。当发送的数据有可能改变 Web 服务器端的数据或者向 CGI 程序传送的数据超过了 1024 字节时（这是 URL 的极限长度），应该使用 POST 方法。如：

```
Post /cgi-bin/main.pl  HTTP/1.1
...
name=zanywoo
```

（3）HEAD

用于浏览器检查 HTML 文档的属性，服务器只返回响应的头标部分内容，而省略被请求文档的内容。

2. 请求头标

（1）Accept

列出浏览器所能够处理的 MIME 数据类型，多个选项之间用逗号分隔，如：

```
Accept: text/html, text/plain
Accept: image/gif
Accept: */*
```

（2）Accept-Charset

列出浏览器所能够支持的字符集，多个选项之间同样用逗号分隔，如：

```
Accept-Charset: iso-8859-5,uicode-1-1
```

（3）Authorization

用于发送带有请求的用户证书。HTTP 协议提供了在非匿名访问时使用的简单询问响应身份验证机制，这个机制使用 MIME base64 编码过程来加密用户名和密码，如：

```
Authorization: Basic asdf345sdf234
```

这里 asdf345sdf234 是已经编码的用户名和密码。

（4）Connection: Keep-Alive

指定使用持久连接(perisistent connection)。如果使用了这个头标，意味着浏览器和服务器将在同一条 TCP 连接上完成多个 HTTP 请求/响应，而不是为每一个请求/响应建立一条 TCP 连接。

（5）Content-Length

指定消息正文内容的长度，如果请求包含消息正文的话。

（6）Content-Type

指定消息正文内容的数据类型，如果请求包含消息正文的话，如：

```
Content-type: application/x-ww-form-urlencoded
```

（7）Date

指出了请求创建的日期和时间。

（8）From

指出了请求用户的电子邮件地址。

（9）Host

指出被访问资源的主机名和端口号。

（10）If-Modified-Since

表示如果所访问的资源在这个时间之后没有被修改过，服务器不需要返回这个资源的内容，浏览器将从本地的缓冲区读取这个资源内容，如：

```
If-Modified-Since: Tue  15 Nov 1998  08:12:23  GMT
```

（11）Referer

指出了浏览器是从哪个页面链接到当前资源的，如：

```
Referer: http://www.somesite.edu.cn/pub/linux/doc/main.html
```

3. 一行空白行

HTTP 协议使用一行空白行来分隔头标部分和消息正文部分，对于没有消息正文的情况，这一空白行就表示请求的结束。注意，这一行不可以省略。

4. 消息正文

如果使用 GET 方法访问资源，没有这部分内容；但是，如果使用 POST 方法来发送表单数据内容，这一部分内容被 URL 编码，并放置在消息正文部分以传送给服务器。请求头标 Content-type 和 Content-length 分别被用来指定使用编码方式和指定这部分数据内容的长度。如：

```
POST /cgi-bin/inforeq.pl HTTP/1.0
...
Content-type: application/x-www-form-urlencoded
Content-Length:556
Name=zanywoo&sex=fE-mail
```

### 15.2.3 HTTP 响应

当服务器处理了 HTTP 请求之后，就返回 HTTP 响应；HTTP 响应包括了以下内容：
状态行
响应头标
一行空白行
消息正文
下面分别介绍如下：

1. 状态行

服务器在 HTTP 响应的状态行中报告了 HTTP 响应的状态，其格式如下：

```
HTTP 版本号 状态号 原因描述
```

如：

```
HTTP/1.0 200 OK
```

HTTP 响应状态号由 3 位数字组成，其中第一位是响应码的分类，如：
2xx 成功，表示被接受，理解和访问；
3xx 更改，表示浏览器将进一步采取动作；
4xx 客户端错误，表示 HTTP 请求中有错误；
5xx 服务器端错误，表示服务器不能执行有效的请求。
上例中的状态号为 200，表示响应状态为成功。

2. 响应头标

响应头标包括以下 8 类：
（1）Allow
列出了服务器所能接受的 HTTP 请求方法，多个选项之间用逗号分隔开。注意，这是在服务器遇到不能处理 HTTP 请求时发送的，如：

```
HTTP/1.0 405 Method not allowed for response
Allow: GET,HEAD,POST
...
```

（2）Content-Type
指出了响应消息正文内容的数据类型。
（3）Content-Length
指出了响应消息正文内容的长度。
（4）Date
响应创建的时间。
（5）Expires
指出了响应的正文内容的过期时间。
（6）Last-modified
指出了当前的正文内容最后一次修改时间。
（7）Location
为浏览器定向导。
（8）Server

3. 一行空白行

这个空白行的作用就是用于分隔响应头标部分和消息正文部分。

4. 消息正文

用户请求的资源内容在响应的消息正文部分传送。

## 15.3 CGI 规范

CGI 是一种接口规范，是操作双方都必须遵守的规则集。它定义了 HTTP 服务器和 CGI 应用程序之间的接口，其中包括命令行、环境变量、输入输出等。

### 15.3.1 CGI 程序工作原理

从 15.1.2 小节的图 15-2 可以看出，访问一个 CGI 程序的完整过程如下：

- 户要求访问一个 CGI 程序资源；
- 浏览器与 HTTP 服务器建立连接，发送 HTTP 请求；
- HTTP 服务器收到请求，并分析浏览器所请求的资源，得知是 CGI 程序；
- 服务器将 HTTP 请求头标设置成环境变量，并执行 CGI 程序；

- 如果请求方法是 GET 方法，则 CGI 程序从环境变量 QUERY_STRING 中获取浏览器发送的数据内容；如果是 POST 方法，则 CGI 程序从标准输入中读取浏览器发送的数据内容；
- CGI 程序处理浏览器发送来的数据，并将处理结果送往标准输出；
- 服务器在 CGI 程序的结果上加上状态行和头标信息，然后将结果返回给浏览器。

一个 CGI 程序要执行的功能可以分成处理输入和处理输出两部分。输入部分是根据请求方法的不同，从环境变量 QUERY_STRING 或标准输入中读取用户输入数据；输出部分是生成 HTTP 响应头标内容和 HTTP 消息正文内容。

### 15.3.2　CGI 方法

读者可能会发现，在 HTTP 协议的请求行里有 GET 请求、POST 请求、HEAD 请求等三种常用的请求方法，它和这里的三种 CGI 方法是一致的，因为 CGI 方法必须要用 HTTP 协议来封装。

CGI 规范的传送方法主要有三种：

- GET 方法；
- POST 方法；
- HEAD 方法。

1. 为 GET 请求读入数据

**程序清单** 15-3：get.pl

```
# ! /usr/bin/Perl
#获取QUERY_METHOD环境变量
$queryType = $ENV{"REQUEST_METHOD"};
print "Content - type: text/plain \n\n";
if($queryType = "GET")
{
      &ReadGetQueryData(*data);
      #打印出读到的数据
      print "The get data is: \n   \n";
      print $data;
      print "\n";
}
#这个子函数的功能就是获取GET请求数据
sub ReadGetQueryData
{
          local(*queryString) = @_  if  @_;
#读取QUERY_STRING环境变量
```

```
$queryString   = $ENV{"QUERY_STRING"
};
return 1;
```

对这个程序的解释：

ReadGetQueryData()函数，它获取一个参数名字，并用用户的输入给它赋值。



在 Perl 语言中，变量是通过“*”操作符使用名字来存取变量的，用户的数据存放在 QUERY_STRING 环境变量中并使用%ENV 关联数组来存取。

2. 为 POST 请求读入数据

**程序清单 15-4：post.pl**

```
#!/usr/bin/Perl
$requestType = $ENV{"REQUEST_METHOD"};
print "Content - type: text/plain \n ";
if($requestType eq "POST")
{
  &ReadPostQueryData(*data);
  print "The Post data is: \n ";
  print $data;
   print "\n";
}
#子函数的功能是获取POST请求数据
sub ReadPostQueryData
{
        local(*queryString) = @_  if @_;
        local($contentLength);
        #读取CONTENT_LENGTH环境变量
        $contentLength   = $ENV{"content_Length"};
        #我们必须首先确认有数据可读
    if($contentLength)
    {
                #从标准输入(stdin)读入 $contentLength长度的字符串，并
                #保存在 $queryString变量中
                read(STDIN,$queryString,$contentLength);
        }
        return 1;
```

```
}
```

对这个程序的解释：

读入数据是通过两个步骤来完成的。首先从%ENV 关联数组中得到数据字符的个数，然后用 read 函数从标准输入中读入这些数据。

### 15.3.3　环境变量

服务器与 CGI 程序交换信息的协作方式是通过环境变量实现的。环境变量是一些保存用户信息的内存区。例如，所有的机器都有一个 PATH 环境变量，如果在当前目录找不到指定的文件时系统便去查找 PATH 变量。当服务器收到一个请求后，它首先要收集它能得到的所有相关信息，并把它放入内存。服务器主要收集如下一些信息：

于服务器自身的详细信息；

关于用户的信息；

关于用户请求的信息。

Perl 提供%ENV 关联数组来操作环境变量。这个数组的关键字（key）是环境变量的名称，值（value）是环境变量的值。

ENV 模块提供了获得环境变量的第二种方法。它从环境变量中生成 Perl 变量，而后读取和设置环境变量就如同处理普通的 Perl 变量一样。

1. 与客户端请求相关的环境变量

表 15-1 列出了与客户端请求相关的环境变量。

表 15-1　　与客户请求有关的环境变量

| 变　量 | 含　义 |
| --- | --- |
| AUTH_TYPE | 服务器所用的确认模式 |
| HTTP_ACCEPT | 这是客户端发给服务器的一个选项的值，其中包括客户端可以接受的数据类型，比如，一个文件浏览器不能将 image/jpg 作为可接受数据类型，而 Navigator 浏览器就可以将 image/jpg 作为可接受数据类型 |
| REMOTE_ADDR | 最终用户的 IP 地址或主机名 |
| REMOTE_HOST | 如果服务器知道客户机的名字，该变量将被置为客户名 |
| REMOTE_IDENT | 如果 HTTP 服务器支持 idend、RFC931 协议，它将被设置为客户端用户名，但是该值不用做验证，只作为登录 |
| REMOTE_USER | 如果用户被确认了的话，则为用户名 |
| CONTENT_TYPE | 含有 CGI 程序的数据文件 |
| CONTENT_LENGTH | POST 请求中向标准输入（STDIN）发送的字节数 |

续表

| 变　量 | 含　义 |
| --- | --- |
| OUTPUT_FILE | CGI 程序放置数据的文件 |
| PATH_INFO | CGI 程序的附加路径信息，它是传送 URL 中程序名后的那一部分内容 |
| PATH_TRANSLATED | PATH_INFO 对应的绝对路径 |
| QUERY_STRING | 传送给 CGI 程序的 URL 的问号（？）之后的那一部分 |
| SCRIPT_NAME | 运行的 CGI 脚本 |



每次服务器收到的请求不可能都是一样的。这些与请求相关的信息包含有用户调用的信息，用户如何发送请求，以及作为请求的一部分传送了多少信息。REQUEST_METHOD、QUERY_STRING、CONTENT_LENGTH 用来表示数据是如何送到 CGI 程序的。

我们先来看一下，Perl 如何读取和设置环境变量。

（1）使用%ENV 的情况

```
#!/usr/bin/Perl
print "\$ENV{'PATH'}= $ENV{'PATH'}\n";
$ENV{'PATH'}  .=  ":/new/path/";
print "\$ENV{'PATH'}= $ENV{'PATH'}\n";
```

对这个程序的解释：

首先 PATH 变量被输出，然后赋值给变量本身，再重新输出。



每当读取%ENV 关联数组时，都将获得环境变量；每当它被设置的时候，环境变量都将被改变。

（2）使用 ENV 模块的情况

```
#!/usr/bin/Perl
use Env qw(PATH);
print "\ $PATH =$PATH \n  ";
$PATH = " :/ande/another/path";
print "\ $PATH =$PATH \n  ";
```

对该程序的解释：

这个程序使用 ENV 模块引入 PATH 变量。



ENV 模块控制$PATH 、$TEMP 变量的使用。当某变量被读取的时候，该模块通过%ENV 获取环境变量，并返回它们结果；当这些变量被赋值的时候，%ENV 数组会被更新。

下面程序说明了 Perl 如何获取客户端发出的 CGI 请求信息：

```
# ! /usr/bin/Perl
print "Content –type : text/plain   \n \n";
    print " Here   are   some   available client information:\n";
print "     HTTP_ACCEPT = ", $ENV{"HTTP_ACCEPT"}, "\n";
print "     REMOTE_HOST = ", $ENV{"REMOTE_HOST"}, "\n";
print "     REMOTE_ADDR = ", $ENV{"REMOTE_ADDR"}, "\n";
print "     REMOTE_USER = ", $ENV{"REMOTE_USER"}, "\n";
print "     AUTH_TYPE = ", $ENV{"AUTH_TYPE"}, "\n";
```

对这个程序的解释：

CGI 程序要求服务器将客户机的信息以环境变量的形式传递给 CGI 程序。程序通过%ENV 获取这些变量来了解客户请求。

2. 和服务器相关的环境变量

表 15-2 列出了和服务器相关的环境变量

表 15-2　　与服务器相关的环境变量

| 变　量 | 含　义 |
|---|---|
| GATEWAY_INTERFACE | 服务器遵守的 CGI 版本 |
| SERVER_NAME | 服务器的 IP 或名字 |
| SERVER_PORT | 主机的端口号 |
| SERVER_SOFTWARE | 服务器软件的名字 |

请看一个具体的例子：

```
# ! /usr/bin/Perl
print "Content –type : text/plain   \n \n";
print " Here   are   some   available client information:\n";
print "    HTTP_ACCEPT = ", $ENV{"HTTP_ACCEPT"}, "\n";
print "    REMOTE_HOST = ", $ENV{"REMOTE_HOST"}, "\n";
print "    REMOTE_ADDR = ", $ENV{"REMOTE_ADDR"}, "\n";
```

```
print "    REMOTE_USER = ", $ENV{"REMOTE_USER"}, "\n";
print "    AUTH_TYPE = ", $ENV{"AUTH_TYPE"}, "\n";
```

对这个程序的解释：

CGI 程序需要 Web 服务器提供有关服务器和服务器所在的计算机的信息，服务器在执行程序前需要设定这些变量。



SERVER_PORT 变量被用在创建动态 HTML，因为动态 HTML 需要指向当前的服务器。客户端默认的 HTTP 端口的值是 80，但是它也不一定是正确的。为了确保正确，在创建动态 HTML 时应使用这个信息，尤其是在生成表格时。

3. 和客户端相关的环境变量

表 15-3 列出了和客户端相关的变量。

表 15-3　　和客户机相关的环境变量

| 变　量 | 含　义 |
|---|---|
| ACCEPT | 列出能被次请求接受的应答方式 |
| ACCEPT_ENCODING | 列出客户机支持的编码方式 |
| ACCEPT_LANGUAGE | 标明客户机可接受语言的 ISO 代码 |
| AUTORIZATION | 标明被证实了的用户 |
| IF_MODIFIED_SINGCE | 当用 get 方式请求并且只有当文档比指定日期更早时才返回数据 |
| PRAGMA | 设定将来要用到的服务器代理 |
| REFFERER | 指出连接到当前文档的 URL |
| USER_AGENT | 标明客户使用的软件 |

# 15.4　CGI 模块

## 15.4.1　使用 CGI 模块

Perl 语言最成功之处在于它有许多第三方的现成的模块可以使用，如 DBI 模块，DBD 模块，CGI 模块等。其中 CGI 模块提供了对 CGI 编程的支持。如下程序：

```
#!/usr/bin/Perl
```

```
use CGI;    #使用CGI模块
$query = new CGI;    #创建一个CGI对象
print  $query->header,    #创建HTTP头标
$query->start_html('Hello World!'),    #开始HTML
$query->h1({-align=>center},'Hello World'),    #一级标题
$query->end_html;    #结束HTML
```

这个例子使用了 Perl CGI 模块来实现最简单的“hello world”程序。

CGI 模块的使用有两种风格。其中一种是面向对象的风格；另外一种是面向函数的风格。使用面向对象的风格时，首先要创建一个 CGI 对象，然后再使用对象的方法，如上面的“hello world”程序就是这种风格。

使用面向函数风格时，则不需要创建 CGI 对象，而可以直接使用 CGI 模块中的函数。如：

```
#!/usr/bin/Perl
use CGI qw/:standard/;    #装载标准CGI模块
print  header,
start_html('hello world!'),
h1('hello Perl cgi world'),
end_html;
```

这里使用的是 CGI 的标准函数，在使用面向函数的风格时，还有如下 CGI 函数集可以使用：

:cgi，所有处理 cgi 的函数，如 param、path_info 等；
:form，所有创建表单内容的函数，textfield；
:html2，所有创建 HTML2.0 标准元素的函数；
:html3，所有创建 HTML3.0 标准元素的函数；
:netscape，所有创建 Netscape 定义的元素；
:html，所有创建 HTML 元素的函数；
:standard，所有标准的函数，是 html2 、html3、 form 和 cgi 的总和；
:all，所有函数。
如：

```
use CGI qw/:netscape/;
use CGI qw/:cgi/;
```

Perl CGI 模块主要提供了三类的函数：创建动态文档、创建 HTML 元素和创建表单。

### 15.4.2 创建动态文档

创建动态文档的方法有以下几类：

- 创建标准 HTTP 头标
- 创建重定向头标
- 创建 HTML 头标
- 结束 HTML

下面分别介绍。

1. 创建标准 HTTP 头标

CGI 程序要生成完整的 HTTP 响应，其中第一部分内容就是 HTTP 头标，它由 header 函数实现，如：

```
header([content_type,status,headers])
```

其中第一个参数 content_type 指定 HTTP 响应正文内容的类型，默认为 text/html；第二个参数是 HTTP 状态和说明消息，默认为“200 OK”，剩余参数是头标中的其他内容，如：

```
print $query->header;
```

或

```
print $query->header('image/gif');
```

或

```
print $query->header('text/html','204 No response');
```

或

```
print $query->header(-type=>'image/gif',
                     -nph=>1,
                     -status=>'402 Payment required',
                     -expires=>'+3d',
                     -cookie=>$cookie,
                     -Cost=>'$2.00');
```

2. 创建重定向头标

有些时候用户不必自己产生文档，只需将浏览器重定向到其他地址，由 redirect 实现，如：

```
print   $query->redirect('http://somewhere.else/in/movie/land');
```

使用 redirect 函数时，其参数应该是绝对 URL 地址，而且，不再需要调用 header 函数来

产生其他头标内容。

有些时候需要指定另外一个参数 nph(no parse header)，如：

```
print $query->redirect(
-url=>'http://somewhere.else/in/movie/land',
-nph=>1);
```

在使用 Microsoft Internet Explorer 时，这个 nph 参数需要设为 1，因为 IE 假定它的脚本都是 NPH(no-parse-header)类型的。

3. 创建 HTML 头标

创建 HTML 文档的第一步就是创建 HTML 头标部分内容。start_html 函数创建一个 HTML 文档的头标部分内容和<BODY>标签，如：

```
print $query->start_html(-title=>'Secrets of the Pyramids',
                         -author=>'fred@capricorn.org',
                         -base=>'true',
                         -target=>'_blank',
                         -meta=>{'keywords'=>'pharaoh secret mummy',
                                 'copyright'=>'copyright 1996 King Tut'},
                         -style=>{'src'=>'/styles/style1.css'},
-BGCOLOR=>'blue');
```

参数 title 设置 HTML 文档的标题；参数 author 设置作者的电子邮件地址，并创建一个<LINK REV = MADE HREF = url>标签；参数 meta 可以设置多个 META 头标行，这个参数是一个散列表，关键字为头标名字（NAME），值为 META 头标的内容（CONTENT）；参数 style 设置样式表内容；其他参数是 BODY 元素的属性（如 BGCOLOR）。

4. 结束 HTML

end_html 函数用来结束 HTML 文档，如：

```
print $query->end_html;
```

这个函数通过输出</BODY> </HTML>来结束 HTML 文档。

### 15.4.3　创建 HTML 元素

CGI 模块可以创建几乎所有的 HTML3.0 和 HTML4.0 元素。CGI 模块不使用普通的函数

来创建 HTML 元素，因此我们将此创建 HTML 元素的方法称为快捷方法(shortcut method)。

HTML 快捷方法的名字与对应的 HTML 元素相同，这些方法返回相应 HTML 标签内容的字符串。可以对这些字符串进行任意处理，如嵌套到其他 HTML 快捷方法中，或者直接输出，如：

```
$q = new CGI;
print $q->blockquote(
                "Many years ago on the island of",
                 $q->a({href=>"http://crete.org/";},"Crete"),
                 "there lived a Minotaur named",
                  $q->strong("Fred."),                    ),
        $q->hr;    #创建一个水平线元素
```

这段代码产生的 HTML 内容如下：

```
<blockquote>
Many years ago on the island of
<a HREF="http://crete.org/";>Crete</a> there lived
a minotaur named <strong>Fred.</strong>
</blockquote>
<HR>
```

如果发现使用这种方法比较烦琐的话，你可以使用面向函数的风格，如：

```
use CGI ':standard';
print blockquote(
        "Many years ago on the island of",
        a({href=>"http://crete.org/";},"Crete"),
        "there lived a minotaur named",
        strong("Fred."),
        ),
      hr;
```

1. 为 HTML 快捷函数传递参数

CGI 模块为很多 HTML 元素提供了快捷方法，如表格（table）。CGI 模块可以创建只有开始标签的元素、有开始标签也有结束标签的元素以及包含属性的函数。

只有开始标签的元素包括<P>、<HR>、<BR>等，其创建方式如下：

```
print $query->hr;
print p;
print $query->br;
```

上面语句分别创建了一个<P>、<HR>、<BR>标签。

有开始标签也有结束标签的元素，称为成对标签，它的内容在快捷函数的字符串参数中指定，如：

```
print h1("chapter 15");
```

生成：

```
<H1> Chapter </H1>
```

CGI 模块还可以传递多个参数，所有这些字符串参数将被组合在一起，参数之间用空格符分开，共同作为标签的内容，如：

```
$name="Anonymous";
print $query->em("Who are you.", $name, "?");
print $query->h3($query->strong($name), ":is your turn.,");
```

分别生成：

```
<em> who are you ,Anonymous ? </em>
<h3> <strong> Anonymous </strong> : is your turn. </h3>
```

CGI 模块在各个参数之间输出空白，这是由变量 （$"） 控制的，如果需要可以改变这种默认得属性，如：

```
{       local($") = '';
        print blockquote(em('Hi'),'mom!'));
}
```

如果需要为 HTML 标签设置属性，可以在第一个参数中传递一个散列表，此表的关键字和值分别对应于 HTML 标签的属性和属性值，如：

```
print a({-href=>'fred.html',-target=>'_new'},
        "Open a new frame");
```

生成：

```
<A HREF="fred.html",TARGET="_new">Open a new frame</A>
```

假如 HTML 标签的属性没有值，那么将对应的值设为 undef，如：

```
print ol({compact=>undef},li('one'),li('two'),li('three'));
```

2. HTML 函数的可分配(distributive)属性

HTML 快捷函数的一个最好的特性是，所有 HTML 标签都是可分配的（distributive）。即如果传递给它们一个数组引用参数，它们将以每个数组项为参数产生 HTML 标签，如：

```
print ul(li({-type=>'disc'},['Sneezy','Doc','Sleepy','Happy']));
```

将生成：

```
<UL>
    <LI TYPE="disc">Sneezy</LI>
    <LI TYPE="disc">Doc</LI>
    <LI TYPE="disc">Sleepy</LI>
    <LI TYPE="disc">Happy</LI>
</UL>
```

利用这个特性，可以方便地生成 HTML 表格，如：

```
print table({-border=>undef},
        caption('When Should You Eat Your Vegetables?'),
        Tr({-align=>CENTER,-valign=>TOP},
        [
            th(['Vegetable', 'Breakfast','Lunch','Dinner']),
            td(['Tomatoes' , 'no', 'yes', 'yes']),
            td(['Broccoli' , 'no', 'no',  'yes']),
            td(['Onions'   , 'yes','yes', 'yes'])
        ]
```

```
        )
    );
```

### 15.4.4　创建表单

Perl 提供的 CGI 模块可以创建任何的 HTML 表单元素。CGI 模块提供的所有函数都会返回相应的 HTML 内容的字符串，而函数调用者只需要负责输出这些字符串。

CGI 模块提供的创建表单元素的函数都会将文本和标签进行 HTML 转义，所以可以直接将一个按钮的标签设置为“<返回>”。

本节要介绍的创建表单元素的函数有：

- 开始和结束表单，startform、endform；
- 文本框，textfield；
- 多行文本框，textarea；
- 密码框，password_field；
- 文件上载域，filefield；
- 弹出式菜单，popup_menu；
- 滚动列表，scrolling_list；
- 复选框，checkbox_group、checkbox；
- 单选按钮，radio_group；
- 提交按钮，submit；
- 取消按钮，reset；
- 隐藏域，hidden；
- 默认按钮，defaults。

1. 开始和结束表单

```
print $query->startform(-method=>$method,
                        -action=>$action,
                        -enctype=>$encoding);
    <... various form stuff ...>
    print $query->endform;
```

或

```
print $query->startform($method,$action,$encoding);
  <... various form stuff ...>
print $query->endform;
```

startform 函数开始一个表单，它返回一个<FORM>标签；endform 函数结束一个表单，同样返回</FORM>标签。

调用 startform 函数时，可以指定请求方法，处理请求的 CGI 程序和表单内容的编码方式。method 参数指定使用的请求方法，如 GET、POST ， 默认是 POST；action 参数指定处理请求的 CGI 程序，默认为当前 URL 值；encoding 参数指定表单数据编码的方式，如 application/x-ww-form-urlencoded、multipart/form-data，默认是 application/x-www-form-urlencoded；CGI 模块在$CGI::URL_ENCODED 变量中保存 application/x-www-form-urlencoded，在 $CGI::MULTIPART 变量中保存 multipart/form-data。

2. 文本框

```
print   $query->textfield(
-name=>'field_name',
                    -default =>'starting value',
                    -size =>50,
                    -maxlength =>80);
```

或

```
print   $query->textfield('field_name','starting value',50,80);
```

textfield 函数创建一个单行文本框，提供输入一行文本的区域。包括有四个参数：name 参数指定文本框的名字；default 为可选参数，指定文本框的默认值；可选参数 size 指定文本框的显示宽度；maxlength 参数指定文本框能接收的最大字符数。

3. 多行文本框

```
print $query->textarea(-name=>'foo',
                    -default=>'starting value',
                      -rows=>10,
                    -columns=>50);
```

或

```
print $query->textarea('foo','starting value',10,50);
```

textarea 函数创建一个多行文本框，提供输入多行文本的区域。它有四个参数：name 参数指定多行文本框的名字；可选参数 default 指定多行文本框的默认值；可选参数 rows 指定

多行文本框的宽度；参数 columns 指定多行文本框的列数。

4. 密码框

```
print $query->password_field(-name=>'secret',
                             -value=>'starting value',
                             -size=>50,
                             -maxlength=>80);
```

或

```
print $query->password_field('secret','starting value',50,80);
```

password_field 函数创建一个密码框，提供输入密码的区域。此函数也有四个参数：name 参数指定密码框的名字；value 是可选参数，指定密码框的默认值；size 参数指定密码框的显示宽度；maxlength 参数指定密码框能接收的最大字符数。

5. 文件上载域

```
print $query->filefield(-name=>'uploaded_file',
                        -default=>'starting value',
                        -size=>50,
                        -maxlength=>80);
```

或

```
print $query->filefield('uploaded_file','starting value',50,80);
```

filefield 函数创建一个文件上载域，提供指定上载文件名的区域。它有四个参数：name 参数指定文件上载域的名字；default 参数指定默认的文件名；size 参数指定文件名输入区域的宽度；maxlength 参数指定文件名输入区域能接收的最大字符数。

如果要提供文件上载域，那么在创建表单时要指定编码方式为 $MULTIPATT(multipart/form-data)，或使用另外一个函数创建表单：start_multipart_form。

在处理表单的时候，可以调用 param 函数获取上载的文件名，如：

```
$filename = $query->param('upload_filename');
```

文件名的格式是浏览器所在操作系统文件名的格式，如 Windows 系统是：

C:\PATH\TO\FILE.MSW，而 Linux 系统为/path/to/the/upload_file。返回的文件名是这个所要上载的文件的文件句柄，可以使用这个文件句柄读取文件的内容，如：

```
while(<$filename>){
    print;
}
```

当一个文件被上载时，浏览器通常要传递一些附加的信息，如 MIME 内容的类等，UploadInfo 函数调用可以获得这些信息。这个函数返回一个包含文件所有信息的散列表，如：

```
$filename = $query->param('uploaded_file');
$type = $query->uploadInfo($filename)->{'Content-Type'};
unless ($type eq 'text/html') {
die "HTML FILES ONLY!";
}
```

上面代码检查所上载的文件类型是否是 text/html。

6. 弹出式菜单

```
print $query->popup_menu('menu_name',
                         ['eenie','meenie','minie'],
                         'meenie');
```

或

```
%labels = (
    'eenie'=>'your first choice',
    'meenie'=>'your second choice',
    'minie'=>'your third choice');
print $query->popup_menu(
                    'menu_name',
                    ['eenie','meenie','minie'],
                    'meenie',\%labels);
```

或

```
print $query->popup_menu(
                        -name=>'menu_name',
```

```
                    -values=>['eenie','meenie','minie'],
                    -default=>'meenie',
                    -labels=>\%labels);
```

popup_menu 函数创建一个弹出式菜单，这是一个 SELECT 元素，其 SIZE 属性 1。Popup_menu 函数的 name 参数指定这个元素的名字；values 参数为数组引用，包含了所有可选的值；default 指定默认选项，缺省为选择第一项；参数 labels 指定选项的显示内容，此参数是一个哈希引用表，其关键字是选项的值，如果不指定这个参数，那么选项显示内容将是选项的值。处理表单时，可以使用 param 函数过得选项的值，如：

```
$os_type = $cgi->param('os');
```

7. 滚动列表

```
print $query->scrolling_list(
                    'list_name',
                    ['eenie','meenie','minie','moe'],
                    ['eenie','moe'],5,'true');
```

或

```
print $query->scrolling_list(
                    'list_name',
                    ['eenie','meenie','minie','moe'],
                    ['eenie','moe'],5,'true',
                    \%labels);
```

或

```
print $query->scrolling_list(
                    -name=>'list_name',
                    -values=>['eenie','meenie','minie','moe'],
                    -default=>['eenie','moe'],
                    -size=>5,
                    -multiple=>'true',
                    -labels=>\%labels);
```

scrolling_list 函数创建一个滚动列表，这是一个 SELECT 元素，但是其 SIZE 属性不为 1。此函数的 name 参数指定这个元素的名字；values 参数是一个数组引用，包含所有可选的值；default 参数指定默认选项，缺省为不选择任何选项；参数 size 指定显示的列表选项的数目；multiple 参数指定是否允许选择多个选项，其值为 true 则允许选择多项，为 false 时只能选择一个选项；lables 选项指定选项的显示内容，此参数是一个哈希表引用，关键字是选项的值，如果不指定这个参数，选项内容将是选项的值。

处理表单时，所有被选择的选项值在一个数组中返回，param 函数可以获得这个数组，如：

```
@selected = $query->param('list_name');
```

8. 复选框

```
print $query->checkbox_group(-name=>'group_name',
                             -values=>['eenie','meenie','minie','moe'],
                             -default=>['eenie','moe'],
                             -linebreak=>'true',
                             -labels=>\%labels);
print $query->checkbox_group('group_name',
                             ['eenie','meenie','minie','moe'],
                             ['eenie','moe'],'true',\%labels);
```

HTML3 兼容的浏览器只能使用如下形式：

```
print $query->checkbox_group(-name=>'group_name',
                               -values=>['eenie','meenie','minie','moe'],
                               -rows=2,-columns=>2);
```

checkbox_group 函数创建一组复选框，这是多个相同名字的、类型为 CHECKBOX 的 INPUT 控件。 name 参数指定这组复选框的名字；values 参数是一个数组引用，包含所有可选项的值；default 参数指定默认选项，可以是单个选项值或者多个选项值的数组引用，分别表示选择一个或多个选项，缺省为不选任何选项；linebreak 参数指定是否在每个选项之间换行，如果此参数值为 true，所有选项为垂直排列，否则为水平排列；labels 参数指定选项的显示内容，此参数是一个散列表引用，关键字是选项的值，如果不指定这个参数，选项显示内容将是选项的值；rows 和 columns 参数指定将所有选项放置在表格中，按行、列排列。

checkbox 函数单独产生一个复选框，如：

```
print $query->checkbox(
        -name=>'checkbox_name',
```

```
-checked=>'checked',
-value=>'ON',
-label=>'CLICK ME');
```

或

```
print $query->checkbox('checkbox_name','checked','ON','CLICK ME');
```

name 参数指定这个复选框的名字；checked 参数指定是否选择这个复选框，当其值为 checked 时，表示默认情况下选择这个复选框；value 参数指定这个复选框的值，如果省略，默认值是 on；label 参数指定复选框的显示内容。

9. 单选按钮

```
print $query->radio_group(-name=>'group_name',
                          -values=>['eenie','meenie','minie'],
                          -default=>'meenie',
                          -linebreak=>'true',
                          -labels=>\%labels);
```

或

```
print $query->radio_group('group_name',['eenie','meenie','minie'],
                          'meenie','true',\%labels);
```

HTML3 兼容的浏览器只能使用如下形式：

```
print $query->radio_group(-name=>'group_name',
                          -values=>['eenie','meenie','minie','moe'],
                          -rows=2,-columns=>2);
```

radio_group 函数创建一组单选按钮，这是多个相同名字的、类型为 RADIO 的 INPUT 控件。name 参数指定这组单选按钮的名字；values 参数是一个数组引用，包含所有选项的值；default 参数指定默认选项，如果省略此参数，则选择第一个选项；linebreak 参数指定是否在每个选项之间换行，如果此参数值为 true，所有选项为垂直排列，否则为水平排列；labels 参数指定选项的显示内容，此参数是一个哈希表引用，关键字是选项的值，如果不指定这个参数，选项显示内容将是选项的值；rows 和 columns 参数指定将所有选项放置在表格中，按行或按列排列。

处理表单的时候，被选择的选项值在一个字符串中返回，可以用 param 函数获得这个选项的值，如：

```
$which_radio_button = $query->param('group_name');
```

10. 提交按钮

```
print $query->submit(-name=>'button_name',
                     -value=>'value');
```

或

```
print $query->submit('button_name','value');
```

submit 函数创建一个提交按钮。name 参数指定按钮的名字；value 参数指定按钮的显示内容。

11. 取消按钮

```
print $query->reset;
```

reset 函数创建一个取消按钮，这是一个类型为 RESET 的 INPUT 控件。

12. 隐藏域

```
print $query->hidden(-name=>'hidden_name',
                     -default=>['value1','value2'...]);
```

或

```
print $query->hidden('hidden_name','value1','value2'...);
```

hidden 函数创建一个不在页面上显示的文本域，这是一个类型为 HIDDEN 的 INPUT 的控件。name 参数指定这个隐藏文本域的名字；default 参数指定它的值，可以是单个值或多个值的数组引用，如果指定多个值，则创建一组相同名字的隐藏域，每个域对应一个值。

param 函数可以返回一个隐藏文本域的值。如果是单值隐藏域，则返回一个标量数据；如果是多值隐藏域，则返回所有这些值组成的数组，如：

```
$hidden_value = $query->param('hidden_name');
```

利用隐藏文本域，可以在会话过程中保存状态信息。所谓会话，指的是需要处理多个页面的一组相关操作。比如在线购物，用户登录之后选择一个或多个所要购买的物品，然后付款。在整个过程中，CGI 程序可以将用户个人信息和购物清单保存在隐藏文本域中，在多个页面中传递这些隐藏文本域。

13. 默认按钮

```
print   $query->defaults('button_label');
```

defaults 函数创建一个按钮，当单击这个按钮的时候，表单页面的所有域的内容都被重新设置成默认值。

14. 综合使用例子

这一节介绍了 CGI 模块提供的函数及其使用方法，下面给出一个综合使用 CGI 模块的例子。代码如下：

**程序清单 15-5：integrate.pl**

```
#!/usr/local/bin/Perl
use CGI;
$query = new CGI;
print   $query->header;
print   $query->start_html("Example CGI.pm Form");
print "<H1> Example CGI.pm Form</H1>\n";
&print_prompt($query);
&do_work($query);
&print_tail;
print $query->end_html;
#子程序print_prompt
sub print_prompt {
        my($query) = @_;
        print $query->startform;
        print "<EM>What's your name?</EM><BR>";
        print $query->textfield('name');
        print $query->checkbox('Not my real name');
        print "<P><EM>Where can you find English Sparrows?</EM><BR>";
```

```
        print $query->checkbox_group(
                                -name=>'Sparrow locations',
                                -values=>[England,France,Spain,Asia,Hoboken],
                                -linebreak=>'yes',
                                -defaults=>[England,Asia]);
        print "<P><EM>How far can they fly?</EM><BR>",
                $query->radio_group(
                        -name=>'how far',
                        -values=>['10 ft','1 mile','10 miles','real far'],
                        -default=>'1 mile');
        print "<P><EM>What's your favorite color?</EM>  ";
        print $query->popup_menu(-name=>'Color',
                                -values=>['black','brown','red','yellow'],
                                -default=>'red');
        print $query->hidden('Reference','Monty Python and the Holy Grail');
        print "<P><EM>What have you got there?</EM><BR>";
        print $query->scrolling_list(
                        -name=>'possessions',
                        -values=>['A Coconut','A Grail','An Icon',
                                'A Sword','A Ticket'],
                        -size=>5,
                        -multiple=>'true');
        print "<P><EM>Any parting comments?</EM><BR>";
        print $query->textarea(-name=>'Comments',
                                -rows=>10,
                                -columns=>50);
        print "<P>",$query->reset;
        print $query->submit('Action','Shout');
        print $query->submit('Action','Scream');
        print $query->endform;
        print "<HR>\n";
}

#子程序do_work
sub do_work {
        my($query) = @_;
        my(@values,$key);
        print "<H2>Here are the current settings in this form</H2>";
```

```
            foreach $key ($query->param) {
                print "<STRONG>$key</STRONG> -> ";
                @values = $query->param($key);
                print join(", ",@values),"<BR>\n";
            }
    }

    #子程序print_tail
    sub print_tail {
            print <<END;
            <HR>
            <ADDRESS>Lincoln D. Stein</ADDRESS><BR>
            <A HREF="/">Home Page</A>
            END
    }
```

## 15.5 计数器实现

计数器可以记录网页被访问的次数，在 Web 上使用的非常普遍。读者可以在许多著名的网站或者是个人主页上可以看到各种各样的计数器。可以有多种方法用来实现计数器，从简单的 SSI 命令到使用 CGI 程序生成内嵌图像等。计数器除了可以记录访问次数外，还可以记录访问者的 IP 地址、操作系统和浏览器的类型等等，使你对访问自己网站的客户情况有一个全面的了解。

### 15.5.1 设计思路

设计计数器程序首先要考虑的问题是计数结果存放在什么地方，是用文本文件还是用 DBM 文件；其次还要考虑并发访问的情况，也就是说用什么方式来保护被并发访问的文件（此文件中保存了计数结果），这可以用文件锁定的方式来实现；最后就是要确定数据的存储格式。

1. 保存计数结果

如果使用 DBM 文件来保存计数结果，那么就可以使用 DBM 数据库提供的函数，如：dbmopen()、dbmclose()、reset()、each()、values()和 keys()。实现计数器程序，只需要使用前两个函数。dbmopen()函数把 DBM 文件与一个关联数组关联起来，其调用方法如下：

```
dbmopen (%hash_name, DB_filename, Read_write_mode);
```

如果指定的数据库文件 DB_filename 不存在，则自动创建两个名为 DB_filename.dir 和 DB_filename.pag 的文件，除非把读写模式设为 undef 值。

在缺省的情况下，当打开 DBM 文件时，只有 64 个记录被读进内存，可以通过给 %hash_name 分配大小来改变此缺省值。如果你只是给自己的网页做计数，缺省值已经足够了，但如果是给整个服务器建立计数器，则可能需要更大的值。

现在看看这三个参数。当调用 dbmopen 时，%hash_name 原有的值都被清除(如果有的话)，以 DBM 文件中的值替换，赋予新值的代码为：$hash_name{'new_key'} = value；当调用 dbmclose （%hash_name）函数时绑定被解除，关联数组中的内容被写入 DBM 文件；当然也可以不关闭文件而将内容写入，方法是调用 reset (%hash_name)函数，注意此函数并不是重置 DBM 文件，而是将内存中的数据写入文件。第二个参数 DB_filename 不包含扩展名。

下面是一个使用 DBM 文件设计计数器的简单例子：

```
dbmopen(%COUNTERS, $DOCUMENT_ROOT/DBM_FILES/counters,0666);
if(!(defined($counters{'my_counter'})){
$counters{'my_counter'}=0;
}
$counters{'my_counter'})++;
$count=$counters{'my_counter'};
dbmclose (counters);
```

也可以使用文本文件来保存计数结果，除了打开、关闭文件外，还要考虑数据的读写问题，同时确定合适的数据格式。整个过程基本步骤如下：

- 打开文件；
- 读取计数；
- 计数加 1；
- 写入新值；
- 关闭文件。

2. 文件锁定

当一个进程在更新文件内容的时候，另外一个进程可能会试图对这个文件进行修改。对计数器程序而言，如果两个或多个人因同时访问页面而调用了计数器程序时，就会出现多个进程同时修改同一文件的情况，这样文件所保存的计数结果就会引起错误。解决办法就是使用文件锁定机制，修改时通知其他试图打开该文件的进程等待，修改完后释放文件，其他进程就可以打开文件并修改。

有两种方法可以实现文件锁定机制：一是创建自己的锁定机制，其次是使用系统函数 flock()。

（1）创建自己的文件锁

这种方法的具体实现是创建和删除一个特定名称的文件，这在资源共享机制中通常称作信号量（semaphore），范例如下：

```
While(-f counter.lock){
    select(undef,undef,undef,0.1);
  }
open(LOCKFILE,">counter.lock);
dbmopen(%COUNTERS, $DOCUMENT_ROOT/DBM_FILES/counters,0666);
if(!(defined($counters{'my_counter'}))){
    $counters{'my_counter'}=0;
  }
$counters{'my_counter'})++;
  $count=$counters{'my_counter'};
dbmclose (counters);
close(LOCKFILE);
unlink(counter.lock);
```

上面的程序首先检查锁定标志文件是否存在，如果存在，就说明另一个进程正在使用该文件，于是等待直到该文件(此处命名为 counter.lock)不存在为止。此处使用了特殊形式循环等待函数 select()，此语句使程序进入休眠状态一段时间，该时间段由最后一个参数定义。这里没有采用 sleep()函数，是因为 sleep()函数基本单位为秒，对这种文件锁定而言太长了，因为几个微秒就足够了。

当锁定标志文件不再存在，就创建自己的锁定标志文件并开始修改计数，完成后关闭该文件并用 unlink 函数将之删除，这样其他的进程又被允许修改计数。锁定标志文件并不是特殊的文件，其文件名可以随意选择。

（2）使用 flock()

flock()函数的调用方法如下：

```
flock (filehandle, lock_type);
```

参数 filehandle 为用 open()函数打开的文件句柄，lock_type 可以为下面四个值之一：

- 定义共享锁。对计数器而言不适用。
- 定义排他锁。
- 定义非阻他锁。此处亦不用。
- 解除锁定。

使用 flock()函数实现的文件锁定范例如下：

```
dbmopen(%counters,"filename", 0666);
# 可以使用不同的文本文件
# OPEN(counters,"<filename")'
flock(counters,2);
if(!(defined($counters{'my_counter'}))){
$counters{'my_counter'}=0;}
$counters{'my_counter'}++;
$count=$counters{'my_counter'};
dbmclose (counters);
flock(counters,8);
```

3. 输出计数结果

最后，需要输出计数结果，Perl 提供有如下三种输出方法：

- 用上面谈到的 SSI 方法输出。
- 创建各种文本格式输出。
- 生成各种漂亮的图形结果输出。

## 15.5.2 代码实现

有了设计思路，计数器的程序实现是很简单的，如程序清单 15-6 所示。

**程序清单 15-6：count1.pl**

```
#!/usr/bin/Perl
#Change Variables
$count = 'count.dat';
##################################################
open (COUNT, "$count")||die("Can't open count.dat\n");
$counter = <COUNT>;
close (COUNT);
open (COUNT, ">$count");
$counter += 1;
print COUNT "$counter";
@indivual = split(//, $counter);
foreach $number (@indivual) {
$output = "$number";
```

```
print $output;
}
close (COUNT);
```

这是一个简单的使用文本文件的计数器程序。当然可以利用图形来实现计数的显示效果，如程序 15-7 所示。

**程序清单 15-7：count2.pl**

```
#!/usr/bin/Perl
#Change Variables
$count = 'count.dat';
$base_url = '\\countpic\\pic\\';
$gif = '.gif';
##########################################
open (COUNT, "$count");
$counter = <COUNT>;
close (COUNT);
open (COUNT, ">$count");
$counter += 1;
print COUNT "$counter";
@indivual = split(//, $counter);
foreach $number (@indivual) {
$output = "<img src=\"$base_url$number$gif\" alt=\"$CNum\">";
print $output;
}
close (COUNT);
```

## 15.6 留言簿实现

### 15.6.1 设计思路

留言簿程序可以用来收集访问者的信息，如访问者的姓名、电子邮件地址、所属公司等，当然还可以有访问者的留言。图 15-4 是一个留言簿程序的界面。

一个典型的留言簿程序主要由三个部分组成：生产留言簿表单的子程序，处理留言簿表单数据的子程序和显示留言簿内容的子程序。

如果留言簿收集的信息需要保存起来，可以使用数据库文件来保存这些信息；也可以使

用普通的文本文件来保存这些信息，每个域之间用特定的分隔符分开。比如说，可以使用冒号“:”作为分隔符，Linux 操作系统的/etc/passwd 文件就使用这种分隔符。

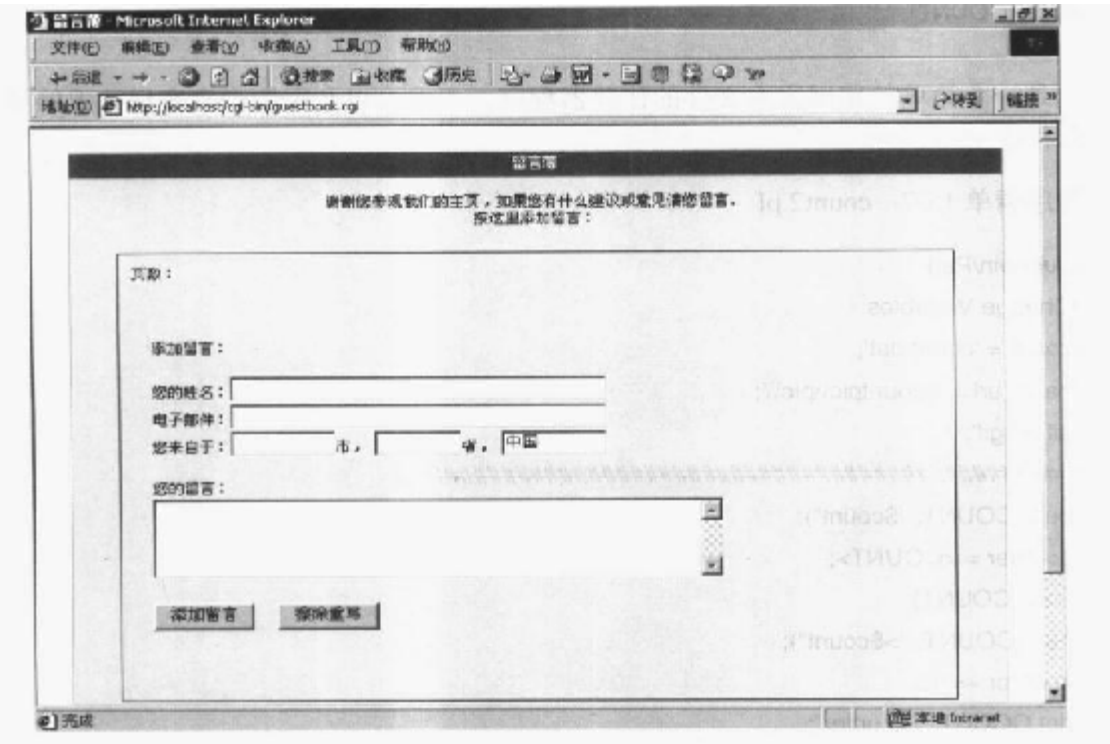



图 15-4　留言簿程序主界面

**程序清单 15-8：guestbook.pl**

```
#!/usr/bin/Perl
### 此程序由zlbox@yeah.net提供，再次表示非常的感谢！！！
###    变量设置
####################
$perpage=15; #设置每页显示留言条数
$masterid="2000"; #管理员密码
$data="data.dat";#设置留言文件路径
##################建立目录#################
unless(-f $data){
open(DATA,">$data")||die $!;
close(DATA);
chmod(0666,$data);
}
##################时间函数#################
```

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday)= localtime(time);
$year=$year+1900;
$mon=$mon+1;
if($mon <10){$mon="0".$mon;}
if($mday<10){$mday="0".$mday;}
if($hour<10){$hour="0".$hour;}
if($min<10){$min="0".$min;}
if($sec<10){$sec="0".$sec;}
$nowtime=$year."-".$mon."-".$mday."/".$hour.":".$min.":".$sec;
##################表单处理#################
if ($ENV{'REQUEST_METHOD'} eq "POST") {
 read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
} else {
 $buffer = $ENV{'QUERY_STRING'};
}
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
   ($name, $value) = split(/=/, $pair);
   $value =~ tr/+/ /;
   $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
   $value =~ s/</&lt;/g;
   $value =~ s/>/&gt;/g;
   $value =~ s/"/&quot;/g;
   $value =~ s/ / /g;
   $FORM{$name} = $value;
   $FORM{'comments'} =~ s/\n/<br>/g;
}
$name = $FORM{'name'};
$E-mail = $FORM{'E-mail'};
$city = $FORM{'city'};
$state = $FORM{'state'};
$country = $FORM{'country'};
$comments = $FORM{'comments'};
$mod = $FORM{'mod'};
$page = $FORM{'page'};
$number = $FORM{'number'};
$id = $FORM{'id'};
##################程序处理###############
```

```
if ($mod eq "add"){&add;} #如果mod=add追加留言
if ($mod eq "dele"){&dele;} #如果mod=dele删除留言
################留言簿界面###############
print "Content-type: text/html\n\n";
print <<EOF;
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>留言簿</title>

<style>
body        { color: #000000; font-family: 宋体; font-size: 9pt }
p         { color: #000000; font-family: 宋体; font-size: 9pt }
br         { color: #000000; font-family: 宋体; font-size: 9pt }
a:link      { color: #000000; font-size: 9pt; text-decoration: none }
a:visited   { color: #000000; font-size: 9pt; text-decoration: none }
a:active    { color: #000000; font-size: 9pt; text-decoration: none }
a:hover     { color: #000000; text-decoration: underline }
</style>

</head>

<body topmargin="0" leftmargin="0">
<div align="center">
 <center>
<table>
 <tr>
  <td width="770" valign="top" height="203">
      <div align="center">
      <table border="0" cellpadding="0" cellspacing="0" width="94%" style="border: 1 solid
#C0C0C0" height="152">
       <tr>
        <td width="100%" bgcolor="#5868C0" height="20">
        <p align="center"><font color="#FFFFFF">留言簿</font></p>
        </td>
       </tr>
 </center>
       <tr>
        <td width="100%" style="border-top: 1 solid #C0C0C0; border-bottom: 1 solid #C0C0C0"
```

```
bgcolor="#DCF0FB" height="112">
    <p align="center"><br>
    谢谢您参观我们的主页，如果您有什么建议或意见请您留言。<br>

    <a href="#add_guestbook">按这里添加留言：</a></p>

      <div align="center">
       <table border="1" cellpadding="10" cellspacing="0" width="90%" bordercolorlight="#000000"
bordercolordark="#DCF0FB">
        <tr>
         <td width="100%" height="101" valign="top">
    <p>
    EOF

    &guestlist; #显示留言

    print <<EOF;
    <form method="POST" action="guestbook.cgi">
     <input type="hidden" name="mod" value="add">
     <input type="hidden" name="username" value="$username">
     <input type="hidden" name="password" value="$password">
     <p><a name="add_guestbook"><br>
     </a></p>
     <div align="center">
      <center>
      <table border="0" cellpadding="15" cellspacing="0" bordercolorlight="#000000"
bordercolordark="#DCF0FB" width="100%">
       <tr>
        <td width="100%" align="left">
         <p align="left">
         <font color="#000000">添加留言：</font><br>
         <br>
     您的姓名：<input type="text" name="name" size="40"><br>
     电子邮件：<input type="text" name="E-mail" size="40"><br>
     您来自于：<input type="text" name="city" size="10">市，  <input type="text" name="state"
    size="8">省，  <input
    type="text" value="中国" name="country" size="10"></p>
     <p align="left">您的留言：<br>
```

```
<textarea name="comments" COLS="60" ROWS="4"></textarea></p>
<p align="left"><input type="submit" value="添加留言">   <input type="reset" value="擦除重
写"> </p>
</td>
</tr>
</table>
</center>
</div>
</form>
<p>
</td>
</tr>
</table>
</div>
<p>
</td>
</tr>
<center>
<tr>
<td width="100%" bgcolor="#000000" height="20">  </td>
</tr>
</table>
</div>
<p>
</td>
</tr>
</table>
</div>
<p>  </p>
</body>
EOF
exit;
#######################显示留言#########################
sub guestlist{
open(DATA,"$data")|| die $!;
@guests=<DATA>;
close(DATA);
$lybnum=@guests;
```

```
$ii=0;
$j=$lybnum;;
while ($j>0) {
$j=$j-$perpage;
$ii++;
}
if($page eq ""||$page<1){$page=1;}
if($page>$ii){$page=$ii;}
$pages=$lybnum-1-($page-1)*$perpage;
for($i=0; $i<$perpage; $i++) {
if($pages-$i>=0){
 $number=$pages-$i;
 $liuyan=@guests[$number];
 $liuyan=~ s/\n//g;
 if ($liuyan ne "") {
  ($lytime,$name,$E-mail,$city,$state,$country,$comments)=split(/\t/,$liuyan);
  if ($state ne "") {$state ="-".$state."省";}
  if ($city ne "") {$city = "-".$city."市";}
  if($id eq $masterid) {$del = "<a
href=\"guestbook.cgi?mod=dele&id=$masterid&number=$number&page=$page\">[删除该条留言]</a>";}
#--------------------------------#
print <<EOF;
留言时间：$lytime $del<br>
留言者：来自$country$state$city的$name<br>
E-MAIL：<a href="mailto:$E-mail">$E-mail</a><br>
留言内容：<br>
$comments
<hr noshade size="1" color="#000000">
<p>
EOF
}
}
}
print "页数：";
$j=$lybnum;
$n=1;
if ($id eq $masterid){$idmod="id=$id&";}
while ($j>0) {
```

```
if ($page != $n){print " <a href=\"guestbook.cgi?".$idmod."page=$n\">[$n]</a>\n";}
 else {print " <font color=\"#C0C0C0\">[$n]</font>\n";}
$j=$j-$perpage;
$n++;
}
}
######################追加留言######################
sub add{
if ($name eq ""){&errorview("请填写您的姓名！ ");exit;}
if ($E-mail eq ""){&errorview("请填写您的E-MAIL地址！ ");exit;}
if ($comments eq ""){&errorview("留言内容不能为空！ ");exit;}
if (!($E-mail=~ /.*\@.*\..*/)){&errorview("您的E-mail输入错误!");exit;}
open(DATA,">>$data")|| die $!;
print DATA "$nowtime\t$name\t$E-mail\t$city\t$state\t$country\t$comments\n";
close(DATA);
}
######################删除留言#################
sub dele{
if ($id ne $masterid){&errorview("对不起，你无权删除留言！ ");exit;}
open(LYB,"$data")||die $!;
@lists=<LYB>;
close(LYB);
open(LYB,">$data")|| die $!;
$n=0;
foreach $list (@lists){
if($n != $number){print LYB $list;}
$n++
}
close(LYB);
print "Content-type: text/html\n\n";
print <<EOF;
<meta http-equiv="refresh" content="0; url=guestbook.cgi?id=$id">
留言成功删除……
EOF
exit;
}
####################errorview##################
sub errorview {
```

```
    print "Content-type: text/html\n\n";
    print "<html><head></head><body><script>alert('$_[0]');history.back();</script></body></html>";
}
```

把这个文件放在 Web 服务器的目录底下，一般情况是/cgi-bin/目录下，然后在浏览器的地址栏中输入“http://localhost/cgi-bin/guestbook.cgi”，就可以运行得到图 15-4 所示的结果了。当然，前题是用户所使用的机器上已经安装了 httpd（http daemon）服务器。

## 15.7 小结与练习

### 15.7.1 小结

本章较为详细地讲解了使用 Perl 进行 CGI 编程的方法。本章首先对 Web 体系结构做了一个简单介绍，使读者对 CGI 编程有一个整体的认识；同时对一个典型的 Perl CGI 程序作了非常详细的分析，使读者对 Perl CGI 程序设计的细节有了较深刻地了解；然后介绍了 HTTP 协议、CGI 规范和 Perl CGI 模块，并通过两个具体的例子：计数器和留言簿的设计使读者不但学习了 CGI 编程本身，同时还提高了如何分析问题和解决问题的能力。

### 15.7.2 习题与思考

（1）设计一个表单，提供两个输入字段，当用户按提交(submit)按钮的时候，把两个字段加在一起。

（2）设计一个 CGI 脚本程序，检测对方的浏览器的状态，并作出响应。

# 第16章

## 数据库编程

数据库基础
SQL 语句
使用 DBI
DBM 数据库
小结与练习



在信息技术迅速发展的今天，信息对人们越来越重要了，因此，对信息的存储和获取也就具有举足轻重的地位。

DBI 是 Perl 编程语言的标准数据库接口。 DBI 独立于数据库，这意味着它可以与任何关系数据库如 Oracle、Sybase、Infomix、Access、MySQL 等协同工作。

本章要点：

- 数据库基础。
- SQL 语句。
- 使用 DBI。
- DBM 数据库。

## 16.1 数据库基础

数据库对其包含的数据有很多种组织方式。最常见的是关系型数据库方式。把采用关系模型的数据库称为关系型数据库管理系统，如当今流行的数据库系统 Oracle、Infomix、Sybase 等都是关系型的典型应用。

我们可以把关系型数据库理解成许多表的集合。表是一个无顺序的行的集合，每一行都有固定长度的字段，每一个字段都可以存储预先确定了类型的数据值，如整型、日期、或字符串。

此外还有面向对象的组织方式。采用面向对象模型，数据库中的一切都被当作某种类的对象。这种类本身定义了操作其封装的数据的规则。这种方式采用面向对象编程语言如 C++、Java、Smalltalk 来实现。Perl DBI 不提供对这种方式的支持。

此外，在不同的操作系统上有很多简单的数据库程序包，这些数据包一般不支持真正的数据库引擎的复杂功能，实际上，它们只是稍微复杂一些的文件处理，而不是真正的数据库程序包，但是，它们的运行是速度非常快，尤其当程序并不需要真正的数据库系统提供复杂功能的时候，使用它们的优点就显得更突出。

以上所讨论的数据库技术，都有两个基本属性。第一个属性是术语的定义：数据库是存储于计算机中的、经过多层抽象过的数据集合，每一层抽象都使存储于其中的数据能够更容易的被组织和存取，这是通过将对某特定数据的请求和获取这个数据的机制分开而实现的。

第二个属性是使用应用程序编程接口（API）提供对数据库中数据的存取。此外，API 还允许用户修改数据库的结构以及存储在数据库中的数据。使用 API 来访问数据库的优点就是用户不必关心 API 如何管理数据，用户实际操作的是经过抽象了的视图。

数据库一般都由多层软件组成，每一层都使用低层定义的接口和服务来实现高层的功能。这种方法具有很多优点。在用户代码和底层数据库之间的抽象允许数据库开发者透明地增加优化、修改数据库文件的结构或者将数据引擎移植到其他平台上，而不需要修改代码。

数据库操作可以分成两种，即对数据本身进行的操作（数据库中逻辑和物理的文件结构）

和对存储在这些文件中的数据进行的操作。前者一般针对特定数据库，可以有很多不同的实现方法，而后者一般是用查询语言实现的。

所有的查询语言，从使用 Perl 字符串和数值型处理功能的最低一层，到高层查询语言如 SQL（Structured Query Language），都实现了可用以操纵数据的四个功能：

（1）取数据

数据库操作中使用最普遍的是检索数据库中的数据。此操作一般是取出数据，然后以查询数据库的 API 宿主语言所能理解的形式返回合适的值。比如，用户使用 Perl 查询一个 Oracle 数据库中的数据，要求使用 SQL 查询语言来取得这些数据，那么返回的行必须是 Perl 字符串和数值的形式。可以使用 SQL 的 SELECT 语句从数据库中取回数据。

（2）存储数据

取出数据之后必然要将数据存储起来，以便后面检索使用。存储管理层将这些数据由编程语言转化成数据库能够理解的值，然后将这些数据存储到数据库文件中。

（3）更新数据

将数据存储到数据库中后，如果需要的话，还可以对它进行修改。比如，一个存储了待售产品信息的数据库，每一个产品的价格信息可能随着时间的改变而改变，对数据库中已有的数据的改变操作称为修改。这个操作既不向数据库中插入新记录也不从中删除记录。

（4）删除数据

对数据进行的最后操作是从数据库中删除任何过时或者冗余数据。这个操作从数据库中将记录永远删除。一旦数据被删除，再也不能恢复了。

## 16.2　SQL 语句

SQL（Structured Query Language，结构化查询语言）语句是为了对存储在数据库中的数据进行操作而设计的。

### 16.2.1　关系型数据库简介

关系型数据库模型围绕着称为表的数据存储单元展开，表中有一系列与之相关的属性，称为列。

在大多数大型数据库系统中，表是在包含有结构信息的模式（schema）创建的。模式是逻辑数据结构或者是模式对象如表和视图的一种集合。在一些数据库中，一个模式与在数据库中创建的一个用户有关，而在另外一些系统中，它是组织相关表的一般方式。

数据在表中是以行（row）的形式来存储的，也就是说数据是存储在含有每一列相应值的一行中的。

可以用 SQL 语句对表中的数据进行操作，SQL 的优越性在于其操纵数据库数据的强大的语法，以及对数据的逻辑结构的增强。

关系型数据库设计的重要一点是，相关信息应该存储在同一个位置，即使存储在不同位置，也应该与原始数据要具有某种意义上的联系。

通过将不同数据分配到不同的表中来避免数据冗余，这种方法称为规范化（normalization）。交叉存储在多个规范化表中的数据，可以通过在表之间建立关联（jion）来进行检索，它允许查询操作从关联设计的表中检索每一个列。但是，在常规基础上创建多表关联，可能会对有大量数据的数据库带来不良影响。在一个表与另外一个表之间进行关联需要更多的磁盘存取，这给数据库的处理带来了困难。

这也是所谓数据仓库的主要原则问题。数据仓库中存储了大量数据，用户要用来输出报表和数据分析。这些问题的一般解决方法是创建更宽的、反规范化了的表来存储与其他表重复的信息。这种方法以存储空间的代价来实现性能的提高，因为存储在数据仓库中的信息一般来说也是只读的，所以不必担心数据改变所带来的同步问题。

### 16.2.2 数据类型

数据库中定义结构（如表和视图）的最重要的一个方面，是每一列的数据类型。SQL 是一种强类型的查询语言，所以，每一个字段或者值的数据类型是给定的，以便用来确定值和字段如何进行比较。

下面来讨论几种常见的数据类型：

（1）数值型

数值数据型包括整型和浮点型，这些类型取决于数据库，其中包括 FLOAT、REAL、INTEGER、及 NUMBER。数值型的比较可用其实际值进行。

（2）字符型

字符型数据用来存放和操纵文本数据。任何字符都可以存放在字符型数据中。注意，如果数字字符以字符型存放，它们将以字符串的相似存放，在排序时是以字符串的形式而不是数字的形式存放，如“10”将会比“7”小。

根据所使用的数据库系统不同，可以有很多种字符型数据，如：CHAR、VARCHAR、VARCHAR2 等等。

对于字符型数据的比较，将根据数据库系统所用的字符集的词典顺序进行。

（3）日期型

在大多数的数据库系统中，至少要实现一种数据类型来包含日期信息，而不使用一个包含日期信息的字符型。这样用户可以非常容易地对日期型数据进行算术运算。

在对日期型数据进行比较时，晚一点的日期比一个早一点的日期要大。存放日期和存放时间戳（日期+时间）的数据类型也是同样。

（4）二进制对象类型

二进制对象类型允许用户在数据库中存储大量无结构的数据，如图像、视频、音频等。实际中不同数据库系统中类型各不相同，但是如果存在的话，一般被称之为 LOB（Large Object）。但是，LOB 类型之间不可以比较。

（5）NULL

NULL 是一个特殊的值，因为实际上它没有值。它可以存在列中，代表给定行中的某一列不包含任何值。在创建表的时候，任何一列都可声明为允许或者不允许 NULL 值，而不管该列的数据类型是什么。

NULL 值和数值 0 是不一样的。0 表示数值 0 本身，而 NULL 表示根本没有数据。

### 16.2.3　创建和删除表

1. 创建表

在讲述具体的 SQL 查询语句之前，先设计两个数据库表 student 和 scores，分别用于存储学生基本信息和学生的学科成绩。

创建表由命令 CREATE TABLE 语句实现，它一般要指定所要创建的表的名字和这个表中各列的定义。

首先创建一个 student 的表，命令如下：

```
CREATE TABLE student (
  id              INTEGER          NOT NULL,
  name             VARCHAR2(32),
  sex             VARCHAR2(8),
  age             INTEGER,
  class           INTEGER,
)
```

然后用 desc student 命令查看（在 Oracle 数据库系统上），输出如下：

```
Name                                      Null?    Type
 ----------------------------------------- -------- ----
ID                                        NOT NULL NUMBER(38)
NAME                                               VARCHAR2(32)
SEX                                                VARCHAR2(8)
AGE                                                NUMBER(38)
CLASS                                              NUMBER(38)
```

使用同样的命令创建 scores 表，此表用于记录学生的成绩信息：

```
CREATE TABLE scores (
id          INTEGER          NOT NULL,
stu_id      INTEGER          NOT NULL,
math        INTEGER(38),
```

```
    english    INTEGER(38),
    chinese    INTEGER(38),
    history    INTEGER(38)
);
```

CREATE TABLE 使用给定的定义创建一个新表，这个表在用户插入数据之前是一个空表。

2. 删除表

使用 DROP TABLE 命令可以删除数据库中的某一个表，以及存储在这个表中的所有数据。比如，要删除 student 表，可用如下命令语句：

```
DROP TABLE student;
```

这样，student 表就被删除，而且是永远删除。

### 16.2.4 插入数据

插入数据的操作应用于用户希望将新记录信息加入到数据库中去。在关系数据库模型中进行数据插入，是在一行一行的基础上完成的。加载到数据库中的每一条记录或者信息都对应表中一个新行，因为每一行插入的记录都是该表中的一个新行。

使用 INSERT 命令将新的记录插入到数据库中，如：

```
INSERT INTO student VALUES(
        1,'zanywoo','male', 23, 1
)
```

INSERT 命令可以指定将数据插入到哪些列中，没有指定的列将是默认值，一般为 NULL。比如，如果只指定 id 字段和 sex 字段，而其他字段允许为 NULL 值，那么 SQL 命令如下：

```
INSERT INTO student (id,sex) VALUES (2,'female');
```

此 SQL 语句在列号和列值之间必须有一个准确的对应关系，同时，给定的值与相关的列之间类型应当匹配。

最后使用同样的 INSERT 命令往 student 表和 scores 表中插入数据。这里构建的表将为后面的试验所用。

### 16.2.5　数据查询

所有的 SQL 查询，不管是简单的还是复杂的，都使用 SELECT 关键字指定要获取的列、所涉及的表以及进行查询所需要的条件。SELECT 是 DML（Data Manipulation Language，数据操纵语言）命令的一种。

1. 简单查询

最简单的查询是找出表中所有行的某些列，其语法如下：

```
SELECT column, column, ... ,column
FROM table
```

或者是

```
SELECT * FROM table
```

这条语句的作用是返回指定的数据库表中的所有列。对于 student 表，使用如下语句：

```
SELECT * FROM student;
```

将输出如下信息：

```
ID NAME                          SEX             AGE        CLASS
- ------------------------------ -------- ---------
1 zanywoo                        male            23           1
2                                female
3 maxnmow                        male            23           2
4 huangchen                      female          20           5
5 meimei                         female          19           6
```

同样可以对指定的列进行查询，如：

```
SELECT id, name, sex FROM student;
```

输出结果如下：

```
ID NAME                          SEX
---------- ------------------------------ --------
```

```
1 zanywoo                male
2                        female
3 maxnmow                male
4 huangchen              female
5 meimei                 female
```

2. 指定查询条件

可以使用 WHERE 命令指定查询的条件。如：

```
SELECT id, name
FROM student
WHERE name LIKE '%zany%';
```

输出结果如下：

```
ID NAME
- ----------------------------
1 zanywoo
```

在返回的信息中只显示了 name 值中含有字符串“zany”的行。WHERE 关键字指定了每一个要返回的行必须满足的条件。

SQL 条件测试语句的比较操作符有：

（1）=

“=”操作符精确比较指定的字段和给定文本的值是否相等，如：

SELECT id,name,sex FROM student WHERE name = 'zanywoo';

将返回 name 列的值等于 zanywoo 的所有行。

（2）<>

和“=”相反，此操作符精确比较列和给定文本的值是否不相等，如：

```
SELECT id,name,sex FROM student WHERE name <> 'zanywoo';
```

将返回 name 列的值不等于 zanywoo 的所有行。

（3）> 和 <

这两个操作符比较指定的列和给定的值，是“大于”还是“小于”的关系，如：

```
SELECT name, sex, age FROM student
WHERE id<4   AND id>2
```

将返回 id 值在 4 和 2 之间（不包括 2 和 4）的学生记录的 name、sex 和 age 列。和“>”和“<”相关的另外两个操作符“<=”和“>=”分别表示“小于等于”和“大于等于”。

（4）IN

此操作符测试指定的字段（列）是否在给定的集合中，如：

```
SELECT name ,sex, FROM student
WHERE name IN ('zanywoo','meimei','huanchen')
```

执行上面语句时，将使用相等操作符对指定的列和集合中的每一个值进行比较，因此，将返回 name 列中包含 zanywoo、meimei 或 huanchen 的行。

（5）LIKE

此操作符允许指定的列和给定的条件值之间进行有限制的匹配，如：

```
SELECT name, sex, class FROM student
WHERE name LIKE '%zany%'
```

将返回 name 列中任意位置包含“zany”的 name、 sex 和 class 列。



注意，字符串中的字符的有限匹配可以通过使用%符号来匹配多个字符，或者是用下划线（_）来匹配单个字符。

可以使用逻辑操作符将测试的条件连接在一起组成多条件列表。SQL 所定义的用来将条件字句连接在一起的逻辑操作符有如下三个：

（1）AND

逻辑与。

（1）OR

逻辑或。

（2）NOT

对后面的逻辑值取反。

在结合使用不同的操作符时，要注意它们的运算顺序。SQL 标准规定，NOT 优先级最高，其次是 AND，然后是 OR，当然可以用括号来改变多个操作符的优先级。

此外，还有一种指定条件语句的方式，即用一个值与查询提供的值来进行比较，如：

```
SELECT name, sex, class FROM student
WHERE name IN
    ( SELECT stu_name FROM stu_table )
```

这是一种频繁使用的指定条件语句的方法。

3. 多表查询

关系数据库的另一个优势在于其进行数据库连接的能力。将存储在不同表中的记录连接在一起，以减少冗余数据，这种记录的连接是关系数据库的关键。

使用多表查询语句可以从多个表中查询指定的列，如：

```
SELECT student.id, student.name, scores.id
FROM student, scores;
```

将返回如下结果：

```
ID NAME                           ID
-- ------------------------------ ----------
 1 zanywoo                        1
 2                                1
 3 maxnmow                        1
 4 huangchen                      1
 5 meimei                         1
```

注意，这里使用了“.”符号，如果不使用的话将会引起混淆。但是，要把表名全部都输出的话，比较麻烦。这时可以使用别名来减少这一麻烦，如：

```
SELECT stu.id, stu.name, sco.id
FROM student stu, scores sco
```

4. 数据分组和排序

通过 ORDER BY 语句可实现对查询得到的数据进行排序。ORDER BY 字句是通过用逗号隔开的列表来指定的。这些列可以对数据进行排序，如：

```
ORDER BY name, id
```

该行代码将用 name 对行进行排序，如果学生记录的名字相同，id 将作为第二个排序键。用户如需将缺省的升序（ascending）排列方向改为降序（descending），只需要在 ORDER BY 语句的任意字段后面添加一个 DESC 关键字。

使用 GROUP BY 语句，用户可以将行进行分组，使其共享一组值，并可对其使用组函数。下面一个例子是要汇总一个表中包含在一个列中的值。在这种情况下，用户可以使用 sun() 分组函数以下面的方式来计算一个给定日期中订货的总值：

```
SELECT order_date, sum(net),sum(vat),sum(total)
FROM sales
GROUP by order_data
```

分组可以逗号分隔列表的形式链接在一起，以创建列的复杂子组。

### 16.2.6　修改数据

1. 使 INSERT 进行数据传输

前面讲了使用 INSERT 语句进行数据插入的操作，INSERT 命令还可以将数据从一个表中传递到另外一个表中。如：

```
INSERT INTO student_copy
    SELECT * FROM student;
```

此语句将 SELECT 命令返回的每一行数据传递到一个新表中，一行一行地插入，直到创建一个完整的复制。注意，表 student_copy 和表 student 必须要有相同的结构。

当然还可以对这条操作进一步细化，如指定在插入之前这些行必须满足的条件，如：

```
INSERT INTO student_copy
    SELECT * FROM student
    WHERE name LIKE '%an%'
```

此外，还可以通过手工指定新表中的目标列，实现从源表的数据析取，如：

```
INSERT INTO stu_name_sex
    SELECT name, sex
    FROM student
```

这里表 stu_name_sex 只有两列，分别是 name 和 sex。

最后，可以从多个表中选择数据来进行插入。如：一个新表 stu_engscore 从 student 表和 scores 表中析取数据，表 stu_engscores 只有三列，语句如下：

```
INSERT INTO   stu_engscore
    SELECT student.id, student.name, scores.English
    FROM student , scores
   WHERE student.id = scores.stu_id
```

2. 删除数据

使用 DELETE 语句可以实现对数据的删除。如：

```
DELETE FROM student
WHERE name LIKE '%zany%'
```

在此，我们同样可以为 DELETE 指定更为复杂的删除条件。

3. 更新数据

通过更新特定行特定列的值可以实现对数据的修改。使用 UPDATE 语句，表中数据的行既不被插入也不被删除，表的结构也没有修改。如：

```
UPDATE student
SET name = 'zhengwu'
WHERE name = 'zanywoo'
```



注意，如果没有指定 WHERE 条件，那么表中的每一行都会被 UPDATE 语句更新。

可以在同一语句中更新多列，如：

```
UPDATE student
SET name = 'zhengwu',
    age = 25
WHERE name = 'zanywoo' AND id=1
```

# 16.3 使用 DBI

## 16.3.1 DBI 与 DBD

DBI 定义了实际的 DBI 编程接口，是面向数据库驱动程序的程序流程和方法调用，并提供了对各种数据库驱动的支持服务。数据库驱动程序是为特定数据库设计的，它负责完成实际的数据库操作。

DBI 与数据库驱动程序的分离使 DBI 成为一个功能强大的编程接口。DBI 可以支持几乎

所有的数据库，如 Oracle、Infomix、MySQL、Sybase、DB2、Empress、SearchServer、PostgreSQL 等。

DBD（Database Driver，数据库驱动程序）是针对每一种数据库开发的，也就是说每一种数据库的 DBD 是不同的。如 Oracle 使用 DBD::Oracle，Infomix 使用 DBD::Infomix。

DBI 的设计使用了 Perl 的面向对象的特性，所以在程序中初始化 DBI 非常容易，加载 DBI 模块语句如下：

```
Use DBI;
```

图 16-1 表示了 DBI 和 DBD 之间的关系。

### 16.3.2　DBI 编程环境的搭建与测试

1. 安装数据库

以安装 MySQL 数据库为例。

首先，根据不同类型的操作系统取得不同的 MySQL 的软件压缩包，例如对于 PC 机上的 redhat linux 可以下载 rpm 软件包 MySQL-3_22_27-1_i386.rpm 和 MySQL-devel-3_22_27-1_i386.rpm。前者是数据库部分，后者是数据库的开发接口包，DBI 通常需要这个包提供的 API 接口。

安装方法是：

```
$rpm –i  MySQL-3_22_27-1_i386.rpm
$rpm –i  MySQL-devel-3_22_27-1_i386.rpm
```

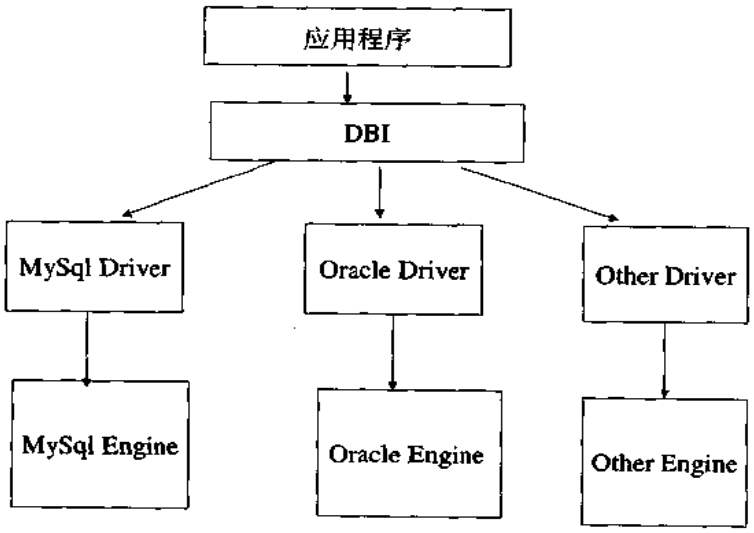



图 16-1　DBI 与 DBD 关系图

2. 安装 DBI

先取得软件 DBI-1_14_tar.tar。
解开压缩包：

```
$gunzip –cd DBI-1_14_tar.tarltar xvf –
```

进入解包后的目录：

```
$cd DBI-1.14
```

然后开始安装：

```
$Perl Makefile.PL
$make
$make test
$make Install
```

3. 安装 DBD

安装 DBD 要根据所使用的不同数据库系统而有所不同。如果使用的是 MySQL，那么就需要 DBD for MySQL 的模块 Msql-Mysql-modules-1.2214.tar.gz；相应地，如果使用的是 Oracle 数据库，那么需要的就是 DBD for Oracle 模块了。
解开压缩包：

```
$gunzip –cd Msql-Mysql-modules-1.2214.tar.gzrltar xvf –
```

进入解包后的目录：

```
$cd Msql-Mysql-modules-1.2208
```

然后开始安装：

```
$Perl Makefile.PL
$make
$make test
$make install
```

4. 设置 MySQL 用户并建立数据库

在开始写程序之前，首先需要对 MySQL 进行设置。因为 MySQL 在刚安装完成的时候，是没有管理员密码也没有用户的，当然还需要建立数据库，以便进行测试。

首先设置数据库的 root 密码。root 是数据库的超级管理员，这里需要注意的是这个 root 与 UNIX 系统管理员 root 毫无关系，它仅仅是数据库的超级管理员。

设置方法是：

```
$mysqi –u root mysqi
```

这时进入提示符“mysql>”然后在该提示符下输入：

```
mysql>update user set password=password('puma') where user='root';
mysql>flush privileges;
```

其中 puma 是数据库超级管理员的密码，也可以是其他的字符串，如 tiger 等等。

然后开始设置用户，假设设置的用户名为 zanywoo 密码为 zanywoo，代码如下：

```
mysql>grant all privileges on *.* to zanywoo@localhost identified by 'zanywoo' with grant option;
```

注意，这里 zanywoo@localhost 表示 zanywoo 这个用户只能在数据库所在的服务器登录才能访问数据库；还可以将它换成 zanywoo@%，这样就表示 zanywoo 这个用户可以在任何地点登录访问该数据库。

“*.*”表示用户 zanywoo 可以访问数据库服务器中的任何对象，all privileges 表示 zanywoo 拥有插入、删除、浏览等所有权限。

最后是建立数据库。可以使用一些图形界面的数据库客户端软件建立数据库，也可以在程序中建立或在 mysql>提示符下使用 sql 语句建立。假设要建立一个数据库名叫 zwdb，其中有一个表叫 zwtable，表中有三个字段 id、name 和 description，其中 id 是整型 int，name，description 是字符型 varchar。

5. 环境的测试

现在就来编写一个测试程序。

**程序清单 16-1：test.cgi**

```
#usr/bin/Perl
use strict;
#使用DBI模块
use DBI;
```

```
#-----连接数据库
$datasrc='mysql:database=zwdb';
$dbh=DBI->connect("dbi:$datasrc",$username,$password) or die DBI->errstr;
$sth=$dbh->prepare("select * from zwtable order by id ");
#----打印出html格式标记
print "<HTML>\n";
print "<head>\n";
print "<meta http-equiv=\"Content-Type\" content=\"text/html charset=gb2312\">\n";
print "<title>产品信息</title>\n";
print "</head>\n";
print "<body>";
#---------循环取出数据并存放在@Data中
while (@Data=$sth->fetchrow_array)
{
      #--------打印出数据
      print $Data[0];
     print $Data[1];
     print $Data[2];;
     print "<br>";
}
print "</body>";
print "</html>";
$dbh->disconnect;
```

程序清单 16-1 存放在服务器的 cgi 目录中，同时执行：chmod 755 testmysql.cgi，然后在命令行下执行：Perl testmysql.cgi ，在正常安装的情况下将可以看到数据库 zwdb 中的所有表名显示出来。当然也可以在浏览器里使用 url 查看结果。

### 16.3.3 句柄

DBI 定义了可以用来与数据库进行交互的三种主要句柄，分别为：

- 驱动器程序句柄；
- 数据库句柄；
- 语句句柄。

#### 1. 驱动程序句柄

驱动程序句柄代表加载的驱动程序，在此驱动程序加载时创建并由 DBI 初始化，每个加载的驱动程序只有一个驱动句柄。驱动程序句柄是 DBI 与数据库驱动程序之间唯一的联系。

和驱动程序句柄相关的函数有两个，一个是 data_source()，此函数列举出可以连接的数据库；另外一个是 connect()，它进行实际的连接。

在 DBI 规范中，一个数据库驱动程序句柄一般用$drh 表示。数据库驱动程序句柄不应当在程序中引用，数据库驱动程序的实例化实际上在 DBI 内部，通常是在 DBI->connect()调用之后。

2. 数据库句柄

数据库句柄封装了一个与特定数据库的连接。在数据库中执行 SQL 语句之前，必须先实际与数据库进行连接，这可以通过 DBI 的 connect()函数来实现，如：

```
$drh = DBI->connect($data_source,...);
```

数据库句柄是相应的数据库驱动程序的子类，支持面向多个数据库类型的多个同步连接，以及面向同一数据库类型的多个同步连接。在 DBI 规范中，数据库句柄一般用$dbh 表示。

3. 语句句柄

语句句柄是 DBI 定义的数据库交互和操作的一种对象，它封装了在数据库中执行的 SQL 语句。

语句句柄是相应的数据库句柄的子类。对于一个给定的数据库句柄，在单个程序中可以创建和执行多个语句句柄，一条语句中的数据是不会被其他语句句柄更改的。在 DBI 规范中，语句句柄一般用$sth 表示。

### 16.3.4　连接和断开连接

其实，数据库编程的绝大多数内容就是在数据库中执行 SQL 语句。但是，在执行语句之前，必须要先建立一个到数据库的连接，当完成数据库操作以后，还要断开和数据库的连接，以释放资源。

1. 连接

在大多数数据库连接时，需要提供如下一些信息：

（1）数据源名

数据源名是一个字符串，它指定了所用的驱动程序、要连接的数据库，或许还有所要连接的数据库的位置等信息。

（2）用户名

用户名指的是要连接到数据库上所用的用户名。有些数据库将数据库划分为几个单独的区域，称为模式（schema），在模式中不同用户可以创建表和操作数据。用户不能够影响其他用户创建的表和数据，但用户可以控制其他用户对自己所创建的表和数据的访问权限。

（3）密码

密码指的是与所提供的用户名相对应的口令。

连接数据库的函数 connect()的定义如下：

```
$dbh = DBI->connect($data_source, $username, $password, \%attr);
```

其中，\%attr 是一个可选参数，它是一个包含应用到这个连接的处理属性，也是一个散列数组的引用，其中最重要的一项是 DBI 是否自动处理错误。如 RaiseError 和 PrintError 属性，当检测到数据库错误时，RaiseError 属性导致 DBI 失败，而 PrintError 属性将自动打印一个警告信息。如果此函数调用成功，将返回一个数据库句柄；如果调用失败，则返回 undef 值。

程序 16-2 演示了如何在程序中连接数据库。

**程序清单 16-2：connectdb.pl**

```
#!/usr/bin/Perl
#
# connect oracle database demo
# 装载DBI模块
use DBI;
# 指定用户名和密码
$user = scott;
$password = tiger;
# 连接数据库
my $dbh = DBI->connect('DBI:Oracle:', $user, $password)
        or die " Can't connect to database". $DBI->errstr;
exit;
```

程序清单 16-2 演示了如何使用 DBI->connect()函数连接数据库，在这个连接中设置了错误处理，当发生错误时，错误信息保存在$DBI->errstr 变量中并被打印出来。

还可以在同一个程序中向同一个数据库连接两次，如程序 16-3 所示。

**程序清单 16-3：connectdb2.pl**

```
#!/usr/bin/Perl
#
# connect oracle database demo
# 装载DBI模块
use DBI;
# 指定用户名和密码
$user = scott;
```

```
$password = tiger;
# 第一次连接数据库
my $dbh1= DBI->connect('DBI:Oracle:', $user, $password)
        or  die " Can't  connect  to  database". $DBI->errstr;
#第二次连接数据库
my $dbh2= DBI->connect('DBI:Oracle:', $user, $password)
        or  die " Can't  connect  to  database". $DBI->errstr;
exit;
```

注意，尽管两次调用 connect()函数时，使用了相同的参数，但是所返回来的句柄却是相互独立的，并没有共享任何信息，所以它们可以独自处理自己所返回来的数据。此外，还可以在同一个程序中连接两个不同类型的数据库，比如一个 Oracle 数据库，而另外一个是 MySQL 数据库。

2. 断开连接

当完成了数据库操作以后，应该显式地断开和数据库的连接，这是一个良好的编程习惯。DBI 模块提供了断开数据库连接的函数，这个函数是和句柄相关的，这在进行多个数据库连接的时候，显得特别有用，此函数定义如下：

```
$rc = $dbh->disconnect();
```

disconnect()函数使用一个指定的数据库句柄进行调用。当有多个数据库句柄处于激活状态时，每一个都必须单独断开。

程序清单 16-4 演示了如何使用 disconnect()函数。

**程序清单 16-4：disconnect.pl**

```
#!/usr/bin/Perl
#
# connect oracle database demo
# 装载DBI模块
use DBI;
# 指定用户名和密码
$user = scott;
$password = tiger;
# 连接数据库
my $dbh = DBI->connect('DBI:Oracle:', $user, $password)
        or  die " Can't  connect  to  database". $DBI->errstr;
```

```
# 断开和数据库的连接
$dbh->disconnect
        or warn "Disconnection failed".$DBI->errstr;
exit;
```

如果不是显式地断开和数据库的连接，那么 Perl 的垃圾收集器将会处理这类断开连接等工作，即当程序中没有任何指向某一个对象的引用，或者是 Perl 要退出时，Perl 将通过对象的 DESTROY 函数来做一些清除工作。

有一点需要注意，如果程序已经修改了数据但是在执行 disconnect()函数之前还没有调用 commit()或者 rollback()等提交函数，不同的数据库将采取不同的行动。比如 Oracle 将自动提交这个修改，而 Infomix 就不这样做。

### 16.3.5 数据库操作

1. 简单查询

使用 Perl DBI 模块从数据库中查询数据，大致需要经过四个阶段：准备阶段，执行阶段，取数据阶段和释放阶段，每一阶段所要完成的工作分别是：准备 SQL 语句，执行所选语句，取数据和释放语句句柄。

（1）准备 SQL 语句

在这一阶段，主要是分析 SQL 语句，对其进行验证，然后返回在数据库中代表这条语句的语句句柄。具体过程是：将 SQL 语句通过一个合法的数据库句柄，以字符串的形式传给数据库；这个字符串由数据库本身进行分析以确保这是一个合法的 SQL 语句；如果数据库对此语句没有报错，将返回某个类型的与数据库相关的数据结构，此数据结构封装了已经分析过了的 SQL 语句；然后 DBI 就使用这个与数据库相关的数据结构来封装语句句柄，最后用户就可以利用语句句柄来完成取数据过程的其他的操作。

这一过程由 DBI 模块提供的 prepare()函数来完成，通过一个数据库句柄来执行这个函数，如程序 16-5 所示。

**程序清单 16-5：prepare.pl**

```
#!/usr/bin/Perl
#
# connect oracle database demo
# 装载DBI模块
use DBI;
# 指定用户名和密码
$user = scott;
$password = tiger;
# 连接数据库
```

```
my $dbh = DBI->connect('DBI:Oracle:', $user, $password)
        or  die " Can't  connect  to  database". $DBI->errstr;
# 创建语句句柄
 my $sth = $dbh->prepare('SELECT * FROM student');
exit;
```

执行 prepare()函数成功，则返回语句句柄，失败则返回 undef 值。

上面使用的 SQL 语句是静态创建的，但是，使用 Perl 的字符串处理功能，还可以临时创建 SQL 语句，并将其传递给 prepare()函数。这在 Web 数据库开发中是一个常用的方法。比如说，如果 student 数据库表放在 Web 上以便于在线浏览，当一个学生想查询信息而键入他自己的名字时，这个包含名字的字符串将被传递给 CGI 程序，然后 SQL 语句用这个字符串从数据库中检索此学生的信息。因此，用户（这位学生）应当能够自己创建 SQL 语句，下面的代码演示了这种方法。

```
# 从在线表上获取变量
my $siteNameTeQuery = $CGI->param("SITE_NAME");
# 引用所获取的变量，以便它可以在SQL语句中使用
my $siteNameToQuery_quoted = $dbh->quote($siteNameToQuery);
# 把引用的变量添加到SQL语句中去
$sth = $dbh->prepare( "
            SELECT name, sex, age
            FROM    student
            WHERE name = $siteNameToQuery_quoted
            ");
$sth->execute();
@row = $sth->fetchrow_array();
# 后面就可以处理所取回来的数据了
```

可见，通过使用 Perl 的字符串处理功能来创建基于用户输入的 SQL 语句，DBI 可以以非常简单和富有弹性的方式来驱动相当复杂的 Web 表单。

（2）执行所选语句

数据查询的第二个阶段是通知数据库执行已经准备好的 SQL 语句，并收集结果数据集。结果数据集指的是用户 SQL 查询所检索回来的数据，SQL 语句是由 prepare()函数成功调用时所创建的语句句柄来执行的，如：

```
# 创建语句句柄
my $sth = $ dbh->prepare('SELECT name,age FROm student');
# 执行SQL语句
```

```
$sth->execute();
```

execute()函数调用成功返回一个 true 值，失败则返回一个 undef 值。如果设置了 PrintError 属性，那么通过 warn()函数调用将产生一个错误消息；如果设置了 RaiseError 属性，那么 die()函数调用将产生一个异常。

（3）取数据

取数据是查询数据的主要阶段。经 SQL 查询检索回来的数据称为结果集，结果集返回到用户的应用程序是以一个一个记录的形式重复完成的。这种基于一行一行的形式来取回结果集的方式称为游标。

游标是用来连续取数据的，记录以它们在结果集中的顺序取回，一般不能跳过记录或进行随机存取，而且游标在结果集中不能后退。所以，从数据库的结果集中重取数据的一般方式是通过语句句柄对结果集中的记录进行循环，一行一行的处理直到所有的行都处理完为止。

对应不同的 Perl 数据类型，可以有很多种方法将数据从结果集中取回。比如说，用户以简单值列表、值数组、字段名/值对或者散列数组等形式取回数据，这些形式都可以从游标检索返回当前行，但以不同的方式返回到用户的 Perl 程序。

Perl 的 DBI 模块提供了许多函数用以从结果集中取回数据，最简单的是 fetchrow_array()函数，这个函数返回一个包含行中字段信息的数组或一个列表，比如从 student 表中取回某个学生的名字和性别，代码如下：

```
# 准备SQL语句
$sth = $dbh->prepare( "
            SELECT name, sex
            FROM   student
            WHERE name = 'zanywoo'
             ");
#执行SQL语句
$sth->execute();
# 从结果集中取回数据
while( ($name,$sex) = $sth->fetchrow_array) {
      print "Student table: name $name sex $sex\n";
}
```

也可以将返回的数据放到一个数组变量中，如：

```
while (@row = $sth->fetchrow_array) {
    print "Student table :name $row[0]   sex $row[1]";
}
```

while 循环一直运行直到括号里的表达式为假时，当没有数据可取时，fetchrow_array()返回一个空列表，Perl 把它当作一个逻辑假值，然后终止此循环。注意，结果集中字段的顺序和 SQL 语句中用户所写的顺序是相同的。

另外一个从数据库的结果集中取数据的函数是 fetchrow_arrayref()，它返回一个数组的引用而不是一个数组，和 fetchrow_array()函数比较，这个函数有更好的性能，因为它取回的行的数据将不用复制到一个新数组中，如：

```
# 从数据库中取回一个数组的引用
while($array_ref = $sth->fetchrow_arrayref){
    print "Student table: name $arrayref->[0]   sex $arrayref->[1]\n";
}
die "Fetch error due to $DBI->errstr";
```

注意，当前语句句柄从数据库中取回的所有行用的都是同一个数组引用，如果要将这些信息存起来以供后面使用，那么程序代码如下所示：

```
# 声明用以保存返回数组行的数组变量$stash
my @stash;
# 取回数组引用
while($array_ref = $sth->fetchrow_arrayref){
    # 复制数组引用所指的值
    push @stash,  [$array_ref];
}
# 输出$stash数组内容
foreach $array_ref    (@stash){
    print "Row: $array_ref\n";
}
```

这里要注意，需要存储的是数组引用所指的值，而不是数组引用本身。

最后一个基于游标的从结果集中取数据的函数是 fetchrow_hashref()，它把数据当作一个散列数组引用。它的使用方法和 fetchrow_arrayref()类似，例如：

```
# 把当前行取回到散列数组引用中
while( $hash_ref = $sth->fetchrow_hashref ){
   # 其他的操作
}
```

引用所指向的散列（哈希）将取回的字段名字作为散列的关键字，将字段的值作为散列

的值。这样，用户可以通过对散列元素的寻址来获取数据库表中的字段，如：

```
# 把当前行取回到散列数组引用中
while( $hash_ref = $sth->fetchrow_hashref ){
    print "Student table :name $hash_ref->{name}   sex $hash_ref->{sex};
}
```

（4）释放语句句柄

当一条语句句柄已经准备好时，所返回的语句句柄与 Perl 程序和数据库服务器上分配的资源是相关联的。当用户不再需要一个语句句柄时，应当销毁它。语句句柄实际上代表的是 Perl 对象，是受 Perl 的垃圾收集器机制约束的。也就是说，当不存在某个语句句柄的引用时，Perl 自己将销毁这个对象，并收回它所占用的资源。

下面看一个例子：

```
if ($fetch_new_data) {
    my $sth = $dbh->prepare(...);
    $sth->execute();
    $data = $sth->fetchall_arrayref();
}
```

这里用户不需要显式地释放语句句柄，Perl 会自动完成这些任务。$sth 变量拥有指向这个特定语句句柄对象的引用。当$sth 变量在这一程序快要结束时消失，最后这个引用也被删除。Perl 的垃圾收集器开始启动。同样的，当程序退出时，所有全局变量消失，并且所有它们所指的对象都以同样的方式被释放。

2. 非 SELECT 语句

在前面讲的 SQL 语句中，除了 SELECT 查询语句，还有插入、删除、更新数据等操作。对于查询操作，是使用游标这个概念对一个结果集进行循环，而对于其他几个操作，情况就不是这样。

对非 SELECT 语句来说，对应于查询操作的四个阶段便显得很麻烦，因为一般用户只需调用这些非 SELECT 语句一次。Perl DBI 模块为这些操作提供了 do()函数，可以通过一个合法的数据库句柄来调用这个函数，如：

```
# 删除表中一个行
$rows = $dbh->do("
        DELETE FROM student
        WHERE name LIKE '%zany%'
      ");
```

如果 do()函数调用成功，则返回一个表示被这条语句影响的函数的值；如果错误，则返回 undef。

使用 do()函数在性能上是有缺陷。如果用户通过重复调用 do()函数来向一个表中插入很多行，可能需要多次准备语句句柄，尤其在语句中包含占位符变量的情况下，更是如此。比如说，向 student 表中插入一些行：

```
foreach $name (qw(smallTiger, axt) ) {
    $dbh->do( "INSERT INTO student (name) VALUES ( ? ) ", undef , $name);
}
```

在数据库内部，对于要插入的每一行，都要创建一个新的语句句柄，准备相应的 SQL 语句，并执行最后销毁。所以，上面这个循环有两个准备调用，两个执行及两个销毁。但是，因为每次循环只需要使用一个绑定值，数据库可能只需要分析这条语句一次就可以了，然后可从共享 SQL 缓存中再次使用这条语句，也就是说，实质上，这个循环浪费了一个准备阶段。

在这种情况下，可以手工创建 SQL 句柄，然后为每一次循环重新执行这个语句句柄，如：

```
$sth = $dbh->prepare("INSERT INTO student (name) VALUES (?)");
foreach $name (qw(smallTiger, axt) ) {
    $sth->execute($name);
}
```

在上面的代码中，只执行了准备语句和两个执行，效率显然比前面一个循环要高多了。

3. 绑定参数

在讲绑定参数的时候，会涉及到若干个概念，如：占位符（placeholder）、参数（parameter）、绑定（binding）等。绑定值是一个可以绑定到 SQL 语句中所声明的占位符的值，但不是将产生的值嵌入到 SQL 语句中。用户指定一个占位符，然后向这个占位符绑定值，如：

```
$sth = $dbh->prepare("
                    SELECT name, sex, age
                    FROM student
                    WHERE name = ?
                ");
$sth->bind_param(1,$siteName);
```

bind_param()函数调用将给定的值与给定的占位符关联起来，数据库会正确地分析占位

符，并且为它预留空间，这个空间将在 bind_param()函数调用时填入。显然，bind_param()函数要在 execute()函数之前调用。

绑定参数可以有两种方法：一种是直接在 execute()函数中绑定，一种是使用 bind_param()函数绑定。

（1）使用 execute()绑定

当调用 execute()函数时，用户可以简单地赋给它一系列值，execute()函数将为用户对其中每一个值调用 bind_param()函数，如：

```
$sth = $dbh->prepare("
                SELECT name, sex, age
                FROM    student
                WHERE name = ? OR sex LIKE ?
              ");
$sth->execute('zanywoo','fe%');
# 其他的操作
```

使用这种方式指定绑定值时，不需要显式地标明这个值的数据类型。大多数情况下，所有值都以 SQL_VARCHAR 类型传递给数据库，但是如果前面用户已经为一些或者是所有占位符显式地指定了一个数据类型来调用 bind_param()，那么将使用这个指定的数据类型，如：

```
$sth = $dbh->prepare("
                SELECT name, sex, age
                FROM    student
                WHERE id= ?
              ");
$sth->bind_param(1,32,SQL_INTEGER);
$sth->execute(2);
```

这样数值 2 将以 SQL_INTEGER 而不是 SQL_VARCHAR 类型作为绑定值，传递给数据库。

（2）使用 bind_param()绑定

使用 bind_param()可以在一个语句中绑定多个值，bind_param()函数使用的索引号从 1 开始一直到要绑定的参数的个数，如：

```
$sth = $dbh->prepare("
        SELECT name, sex, age
        FROM student
        WHERE name LIKE ?
        AND sex LIKE ?
```

```
        ");
$sth->bind_param(1,'zany%');
$sth->bind_param(2,'fe%');
```

最好使用“？”占位符，因为它可以保证不同数据库之间的兼容。

使用绑定值与嵌入的临时 SQL 语句有什么区别呢？嵌入语句的创建使用了 Perl 的字符串处理功能来创建一个完整的 SQL 语句，然后发送到数据库中；而绑定值位于 SQL 语句之后，但在它真正执行之前，发送到数据库中去。这两种情况结果是一样的。但是，真正的差别在于数据库处理绑定值的方式。如多数数据库系统都支持称为共享 SQL 缓存的数据结构，在这个结构中，与 SQL 语句一起存储了一些附加的信息。如果某条语句已经存在于共享 SQL 缓存中，那么数据库中在返回一个指向这条语句之前，不需要重新处理这条语句，它可以简单地重新使用缓存器中存储的信息。这种处理方式，在很大程度上提高了性能。

Perl 是一个类型宽松的语言，为了帮助驱动程序确认绑定值中所提供的数据的类型，用户可以在 bind_param()函数中提供一个附加的参数来指定这个数据类型，如：

```
use  DBI qw(:sql_types);
$sth = $dbh->prepare( "
                SELECT stu.name, stu.sex, stu.class, sco.English
                FROM student stu, scores sco
                WHERE name = ?
                AND id = ?
                AND stu.id = sco.stu_id
            ");
# 绑定参数，不指定类型
$sth->bind_param(1,"zanywoo");
# 绑定参数，并指定类型
$sth->bind_param(2, 1, {TYPE => SQL_VARCHAR} );
# 执行语句句柄
$sth->execute();
```

注意，如果指定了一个数据类型，驱动程序应当将其作为一个重要的指示，但它也仅仅是一个指示，有些驱动程序并不会注意这个类型。

4. 绑定输出列

Perl DBI 模块提供了一些功能用以简化取数据的操作，并且可以避免多余的复制，这是由 bind_col()函数或 bind_column()函数实现的。执行这两个函数的操作称为绑定列，它是通过命名一个可以直接用来存储特定列值的 Perl 变量来进行的。当用 fetch()函数从数据库中取

回数据时，与每一列相关的 Perl 变量将自动用取回的值更新。

下面代码演示了这个过程：

```
# 声明用来存储字段值的Perl变量
my ($name, $sex, $age);
# 准备SQL语句并执行
$sth = $dbh->prepare("
                SELECT name, sex, age
                FROM student
                WHERE name = 'zanywoo'
              ");
$sth->execute();
# 把输出列和Perl变量关联起来
$sth->bind_col(1,\$name);
$sth->bind_col(2,\$sex);
$sth->bind_col(3,\$age);
# 从结果集中取回数据
while ( $sth->fetch ){
    print "student table: name $name   sex $sex    age   $age\n";
}
```

bind_col()函数用以将 Perl 变量和输出的列相关联，这个函数以要关联的列的索引号（从 1 开始）和一个指向要关联的 Perl 变量的引用作为参数。当 fetch()调用完成时，相关联的 Perl 变量将自动更新而不用显式地赋予检索回的值。bind_param()函数是通过一个执行过的语句句柄来调用的。

另外一个绑定输出列的函数是 bind_column()，它的工作方式和 bind_col()差不多，但是它不需要显式地指定 Perl 变量要绑定的列号，而只需要简单的指定那些 Perl 变量，列的赋值将自动发生，如：

```
# 声明用来存储字段值的Perl变量
my ($name, $sex, $age);
# 准备SQL语句并执行
$sth = $dbh->prepare("
                SELECT name, sex, age
                FROM student
                WHERE name = 'zanywoo'
              ");
$sth->execute();
```

```
# 把输出列和Perl变量关联起来
$sth->bind_column(undef, \$name, \$sex, \$age);
# 从结果集中取回数据
while ( $sth->fetch ) {
    print "student table: name $name   sex $sex    age   $age\n";
}
```

注意，在 SQL 语句中指定的列数与在 bind_column()中指定的 Perl 变量数必须准确匹配，用户不可以像使用 bind_col()一样挑选要取回的列。但是，由于 bind_column()函数在内部使用了 bind_col()函数，所以，使用这两个函数的规则是相同的。

5. 检索

数据库的检索操作分为基本检索（stomic fetching）和批量检索（batch fetching），这是从数据库中获取数据的两种方法，它们彼此关联，但是实现的方式不同。

（1）基本检索

前面已经讲过了使用语句句柄来执行 SQL 语句，但是这需要四个步骤来完成，显得很麻烦。而基本检索允许用户将数据检索的四个步骤归并为一个函数来实现，实现这一功能的函数有两个，分别是 selectrow_array()和 selectrow_arrayref()。这两个函数不需要准备和执行语句句柄，而且只返回一行数据。

下面的例子演示了如何使用这个函数：

```
($name, $age) = $dbh->selectrow_array("SELECT name, age
                                       FROM student");
print"student table :name $name age $age\n";
```

这里的 SQL 语句没有给出 WHERE 条件语句，却只返回一行数据，即返回数据库表中的第一行。而且，从这个例子中可以看出，selectrow_array()函数是通过一个合法的数据库句柄来执行的。

当然，selectrow_array()函数也可以使用绑定值，这样有利于数据库资源的重用。

（2）批量检索

fetchrow_array()函数是基于行来查询的，它在结果集中进行遍历。批量检索通过一个函数调用以返回 SQL 查询的整个结果。Perl DBI 模块为实现这个功能，定义了许多函数，如 fetchall_arrayref()和 selectall_arrayref()函数，它们可以将整个结果集存入一个 Perl 数据结构中以供程序使用。

fetchall_array()函数根据传递给它的参数的不同，可以以三种不同模式的操作，这三种模式分别对应于 fetchall_array()函数不带参数调用，以一个数组引用作为参数调用，以及以一个散列引用作为参数调用的三种状态。

当不带参数调用 fetchall_arrayref()时，它将返回一个指向数组的引用，这个数组中包含了结果集中每一行的引用，这些引用都指向包含一行数据的各个字段值的数组。程序清单 16-6 解析了不带参数调用 fetchall_arrayref()返回的数据结构：

**程序清单 16-6：fetchall.pl**

```
#!/usr/bin/Perl
#
# connect oracle database demo
# 装载DBI模块
use DBI;
# 指定用户名和密码
$user = scott;
$password = tiger;
# 连接数据库
my $dbh = DBI->connect('DBI:Oracle:', $user, $password, { RaiseError=>1 } )
          or  die " Can't  connect  to  database". $DBI->errstr;
# 创建语句句柄
my $sth = $dbh->prepare("
                SELECT name, age, sex
                FROM  student
               ");
# 执行SQL语句
$sth->execute();
# 调用fetchall_arrayref()，并把结果集存放在一个Perl数据结构中
my $array_ref = $sth->fetchall_arrayref();
# 输出$array_ref中存储的数据
foreach my $row (@$array_ref) {
    my ($name, $age, $sex) = @$row;
    print "\tStudent table: name $name, age $age, sex $sex\n";
}
exit;
```

很显然，如果用户希望从数据库中检索出所有的结果集，fetchall_arrayref()函数是最好的选择。

当以数组引用作为参数调用 fetchall_arrayref()函数时，可以返回一个包含结果集中每一行的某些列的数据结构。

对应于某一个 SQL 语句：

```
SELECT name, sex, age, class
FROM student
WHERE id = 1
```

那么每一个返回行的数组下标将是：

```
name        -> 0
sex         -> 1
age         -> 2
class       -> 3
```

既然知道了这些列的数组下标，那么就可以指定返回结果集中每一行的某些列了，如：

```
# 指定取name字段和class字段
$array_ref = $sth->fetchall_arrayref([0,3]);
```

数组下标是按照标准 Perl 要求指定的，因此用户也可以使用范围来表示下标值，如：

```
# 指定取从第一列到第三列
$ array_ref = $sth->fetchall_arrayref([0 .. 2]);
```

使用 fetchall_array()函数的第三种模式是传递一个包含要存储列的散列引用参数，有选择地将列存储到一个数组引用中，它返回一个指向包含结果集中所有行的散列引用数组的引用。

比如对于下面的 SQL 语句：

```
SELECT name, age, sex
FROM student
```

为了有选择地存储 name 列和 sex 列，可以让 fetchall_arrayref()通过采用一个匿名散列作为参数来存储相应字段，这个散列必须以包含要存储的列的名字来进行初始化，如：

```
# 指定存储name列和sex列
$array_ref = $sth->fetchall_arrayref( {name=>1, sex=>1} );
```

程序 16-7 演示了如何遍历 fetchall_arrayref()函数调用所创建的数据结构：

程序清单 16-7：fetchall_ds.pl

```
#!/usr/bin/Perl
#
# connect oracle database demo
# 装载DBI模块
use DBI;
# 指定用户名和密码
$user = scott;
$password = tiger;
# 连接数据库
my $dbh = DBI->connect('DBI:Oracle:', $user, $password, { RaiseError=>1 } )
          or  die " Can't  connect  to  database". $DBI->errstr;
# 创建语句句柄
my $sth = $dbh->prepare("
              SELECT name, age, sex
              FROM  student
            ");
# 执行SQL语句
$sth->execute();
# 调用fetchall_arrayref()，并把结果集存放在一个Perl数据结构中
my $array_ref = $sth->fetchall_arrayref( {name=>1, sex=>1} );
# 遍历fetchall_arrayref()所创建的数据结构
foreach my $row (@$array_ref) {
    print "\tStudent table: name $row->{name}, sex $row->{sex}\n";
}
exit;
```



使用 fetchall_arrayref()时需要注意，返回的散列中列名总是小写，忽略了数据库使用的大小写和 fetchall_arrayref()参数的大小写。

## 16.4 DBM 数据库

大多数的 Linux 系统或者是 UNIX 系统，都有一种叫做 DBM 的数据库。它提供有简单的数据库管理工具，程序可以把键/值对存储到磁盘文件中。这些文件保留了数据库中的数值，

这样，程序就可以使用该数据库。同时，这些程序还可以添加新的数值，更改已有的数值，或者是删除旧的数值。

DBM 数据库以文件的方式储存数据，由于采用散列结构进行连接，因此具有一些特殊功能。与普通文本数据库相比，它具有稳定，检索速度快和支持量大的优点。但由于 DBM 是从 UNIX 系统 中移植来的，因此在 Vnix/linux 系统中优点比较明显，而在其他系统如 NT 系统中则不太理想。

Perl 使用散列数组来处理 DBM 数据库，当打开一个 DBM 数据库时，将一个散列数组与这个 DBM 数据库相关联；然后使用这个散列数组去访问这个 DBM 数据库的内容。在此散列数组中创建一个新的元素，相当于在 DBM 数据库中添加一个新的记录；从这个散列数组中删除一个元素，相当于从 DBM 数据库中删除这个值。

Perl 提供了许多种 DBM 数据库的程序包：

AnyDBM_File —— 这是一个为其他 DBM 包所继承的虚类，提供一个标准的 DBM 接口。

GDBM_File —— GNU 版本的 DBM 数据库接口。

NDBM_File —— Berkeley VNIX 版本的 DBM 数据库接口。

ODBM_File —— Perl 标准数据库包。

SDBM_File —— Perl 标准数据库包。

DB_File —— Berkekey DB 接口。

AnyDBM_File 包封装了其他的包，如果用户使用 AnyDBM_File 包，程序将自动加载上面几个包中的一个，当然，也可以指定加载一个 DBM 包，如：

```
use AnyDBM_File;     # 程序将自动加载其他DBM包中的某一个包
use SDBM_File;       # 用户可以指定加载一个DBM包
```

### 16.4.1　创建和打开 DBM 文件

操作 DBM 数据库之前，必须先打开这个数据库，并将这个数据库与一个散列数组相关联，可以用下列两种方法实现：

1. 使用 dbmopen()函数

dbmopen()函数调用方式如下所示：

```
dbmopen(%hash, "dbfilename",$mode);
```

其中，%hash 参数是一个 Perl 散列数组变量，函数返回后，将此散列数组与数据库相关联。dbfilename 参数是 DBM 数据库文件名；$mode 参数是打开数据库文件的模式，如 $mode=0644，表示所有者拥有读写权限，组用户拥有读权限，其他用户拥有读权限。如果指

定的数据库文件不存在，则使用这个模式创建这个数据库；如果这个数据库已经存在，则此模式不起作用。最后使用 dbmclose()函数关闭数据库文件。

2. 使用 tie()和 untie()函数

tie()函数调用方式如下所示：

```
use AnyDBM_File;
# 打开数据库文件，并与%hash变量关联，如果文件不存在，则创建
tie %hash, "SDBM_File", $filename
or die " Can't open $filename: $!\n";
# 其他的操作。
untie %hash;      # 关闭文件
```

对于不同的 DBM 包，tie()函数的定义不尽相同，下面是 DB_File 包的 tie()函数原型：

```
tie %hash,'DB_File',$FileName, $flags,$mode,$DB_HASH;
```

其中%hash 参数是要与指定数据库文件关联的散列数组；DB_File 参数标明数据库类型是 Berkelay DB；$FileName 参数是数据库文件名；$flags 参数可以是 O_RDWR（数据库可读写）、O_CREATE（数据库可创建）或者是两者的结合；$mode 参数设置了数据库文件的权限，和 dbmopen()函数的对应参数一样；$DB_HASH 参数指定数据库的类型是散列数值形式的不规则数据库，类似的，DB_BTREE 表示可排序的对称二叉树类型，DB_RECO 为记录行类型。

## 16.4.2 插入和删除记录

创建或打开了 DBM 数据库以后，所有的操作均使用这个与数据库相关联的散列数组，在散列数组中添加或删除一个记录，将立即更新对应的数据库文件内容。

1. 插入记录

下面代码向数据库文件中插入一个姓名为“zanywoo”，电话号码为“12345678”的记录：

```
use AnyDBM_File;
# 打开数据库文件，与%hash数值关联，如果文件不存在，就创建
tie %hash, "DB_FILE", $FileName     or die " Can't open $FileName: $!\n";
# 增加一条记录
%hash{"zanywoo"} = "12345678";
untie %hash;
```

在散列数组中增加记录，就是给一个新的关键字赋值，如果该关键字存在，则修改原内容。当然，更为健壮的程序需要确认该关键字是否存在于散列数组中，如：

```
use AnyDBM_File;
# 打开数据库文件，与%hash数值关联，如果文件不存在，就创建
tie %hash, "DB_FILE", $FileName     or die " Can't open $FileName: $!\n";
if (exists $hash->{"zanywoo"})  #   判断关键字是否存在
{
    print "error: zanywoo have already exist in the database. \n";
}
else{
    # 增加一条记录
    %hash{"zanywoo"} = "12345678";
}
untie %hash;
```

2. 删除记录

通过删除与数据库文件关联的散列数组中的数据项，就可以实现数据库文件记录的删除。使用 delete()函数删除散列数组中数据项的方法，如下：

```
use AnyDBM_File;
# 打开数据库文件，与%hash数值关联，如果文件不存在，就创建
tie %hash, "DB_FILE", $FileName    or dle " Can't open $FileName: $!\n";
# 删除一条记录
delete ($hash{"zanywoo"};
untie %hash;s
```

## 16.4.3　遍历 DBM 数据库

可以通过遍历与数据库文件关联的散列数组实现对 DBM 数据库的遍历，如：

```
use AnyDBM_File;
# 打开数据库文件，与%hash数值关联，如果文件不存在，就创建
tie %hash, "DB_FILE", $FileName    or die " Can't open $FileName: $!\n";
foreach $key (keys %hash) {
    print   "$key的电话号码是  $hash{$key}\n";
}
```

```
# 关闭DBM数据库
untie %hash;
```

这里，foreach 循环扫描数据库两次，第一次是访问所有的关键字，第二次是访问关键字对应的值，为了提高程序的效率，可以使用如下的方式：

```
use AnyDBM_File;
# 打开数据库文件，与%hash数值关联，如果文件不存在，就创建
tie %hash, "DB_FILE", $FileName   or die " Can't open $FileName: $!\n";
while ( ($key,$val) = each(%hash) ) {
    print   "$key的电话号码是  $val\n";
}
# 关闭DBM数据库
untie %hash;
```

此外，还可以利用 Perl 强大的字符串处理功能，进行查询条件的匹配。例如，用户可以查询姓名中包含“zany”字符串的记录，如程序清单 16-8 所示：

**程序清单 16-8：dbm.pl**

```
use AnyDBM_File;
# 打开数据库文件，与%hash数值关联，如果文件不存在，就创建
tie %hash, "DB_FILE", $FileName   or die " Can't open $FileName: $!\n";
# 请输入要查找的字符串
$searchfor = <STDIN>;
chop $searchfor;
$I = 0;
while ( ($key,$val) = each(%hash) ) {
    if ($key = ~/$searchfor/) {
        $i++;
        print   "$I:     $key的电话号码是  $val\n";
    }
}
# 关闭DBM数据库
untie %hash;
```

从上面这些程序中可以看出，DBM 数据库的使用确实很简单，可以用 DBM 文件代替一般的文本文件来保存信息，这样程序的实现会变得更加容易，代码也更加清晰。

## 16.5　小结与练习

### 16.5.1　小结

本章通过具体的例子讲解了数据库的操作，内容包括对 SQL 语句、DBI 编程接口和 DBM 数据库的详细解释。通过本章的学习，读者能够知道如何在 Perl 中使用 DBI 编程接口来实现对 Oracle、MySQL 等大型数据库的操作，同时还可以学会如何使用一种简单、有效且使用广泛的 DBM 数据库。其中本章对 DBI 的句柄介绍将帮助读者对 DBI 的使用有深刻的认识。

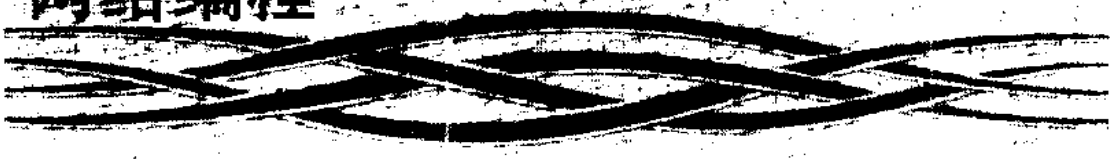
### 16.5.2　习题与思考

（1）写两个程序：一个由“<>”读数据，以 word 为单位进行切割，随后用 DBM 文件记录每个 word 出现的次数。另外一个程序由 DBM 文件读入记录数据，打印出所有的 word，按出现次数由大到小进行排列。

（2）写一个程序，打开 sendmail 的 aliases 数据库，打印出所有字段的内容。

# 第17章

## 网络编程



Socket 基础
Perl 对 Socket 的支持
面向连接模型应用
使用预编译模块
RPC 的实现
小结与练习



10 年以前，我们还很少有人接触过网络。现在，计算机通信已成为我们社会结色的一个基本组成部分。网络被应用于工商业各行各业的各个方面，网络应用覆盖了政府部门、教育部门、工商企业甚至家庭。简而言之，计算机网络已遍布各个领域。全球因特网的持续发展是网络领域最令人感兴趣的现象之一。10 年前，因特网仅是一个只有几十个站点的研究项目。今天的因特网已成为一个连接几乎所有国家的通信系统。数据网络使个人化的远程通信成为可能，并改变了商业通信的模式。一个完整的用于发展网络技术、网络产品和网络服务的新兴工业已经形成。计算机网络的普及性和重要性刺激了不同网位上对具有更多网络知识人才的大量需求。企业需要雇员规划、获取、安装、操作、管理那些构成计算机网络和因特网的软硬件系统。另外，计算机编程已不再局限于个人计算机，而要求程序员设计并实现能与其他计算机上的程序进行通信的应用软件。

Perl 是一门强大的编程语言，通过 Perl 提供的各个模块，你可以很容易地进行不同领域的程序开发，包括前面讲过的 CGI 编程，数据库编程，还有我们将要在这一章涉及到的网络编程。不过这里有一点要说明，有人把利用 CGI 进行的 Web 编程也看作为网络编程，但是，这里要讲的主要是在许多应用软件开发中用的最多的 Socket 技术。

本章要点：

- Socket 基础。
- Perl 对 Socket 的支持。
- 面向连接模型应用。
- 使用预编译模块。
- RPC 的实现。

# 17.1 Socket 基础

## 17.1.1 Socket 原理

Socket 编程界面首先由 4BSD VNIX 提出，目的是解决网间网的通信问题。

Socket，其英文原意是“插座”的意思。其实在日常生活中，也有插座，比如信号插座，电源插座，有的插座可以接收信号，有的可以发送信号。假如在电话线和电话机之间安放一个插座（这相当于二者之间的接口，这一部分装置在物理上是存在的，不过往往不是以具体的形式表现出来，而是两根简单的接线柱而已），则 Socket 地作用与这个电话插座非常相似。

将电话系统和面向连接的 Socket 机制相比，它们非常的相似。以一个国家级的电话网为例，电话的通话双方相当于通信的两个进程，通话双方所在的地区相当于一个网络，区号是它的网络地址；区内一个单位的交换机相当于一台主机，主机分配给每一个用户的局内号码相当于 Socket 号。

任何一个用户在通话之前，首先要拥有有一部电话机，也就是相当于申请一个 Socket 号；同时要知道对方的号码，相当于对方有一个固定的 Socket。然后向对方拨号呼叫，相当于发


http://www.ypcnet.com

出连接请求（假如对方不再同一个区内，还要拨对方区号，相当于网络地址）。对方假如空闲（相当于通信的另一主机且可以接收连接请求）并拿起了电话话筒，双方就可以正式通话了，相当于连接成功。

双方通话的过程，是向电话机发出信号和电话机接收信号的过程，相当于向 Socket 发送数据和从 Socket 接收数据。通话结束以后，一方挂起电话机，相当于关闭 Socket，撤销连接。

在电话系统中，一般用户只能感受到本地电话机和对方电话号码的存在，建立通话的过程，话音传输的过程以及整个电话系统的技术细节都是忽略的，这也与 Socket 机制非常的相似。Socket 利用网间网通信设施实现进程通信，但是它对通信设施的细节毫不关心，只要通信设施能够提供足够的通信能力，它就满足了。

到这里为止，读者已对 Socket 有了直观的了解。抽象地说，Socket 实际上提供了进程通信的端点。进程通信之前，双方首先必须各自创建一个端点，否则没有办法建立连接并相互通信。

在 Internet 中，每一个 Socket 使用一个半相关描述，如：

```
{ 协议,本地地址,本地端口}
```

一个完整的 Socket 连接则用一个相关描述：

```
{ 协议,本地地址,本地端口，远程地址，远程端口}
```

每一个 Socket 有一个本地唯一的 Socket 号，由操作系统分配。

必须明白，Socket 是面向 client/server 模型而设计的，针对客户和服务器程序提供不同的 Socket 系统调用。客户机申请一个 Socket，系统为之分配一个 Socket 号；服务器拥有全局公认的 Socket，任何客户都可以向它发出连接请求和信息请求。



Socket 利用客户——服务器模式巧妙地解决了进程之间建立通信连接的问题。服务器端半相关描述为全局公认非常重要。假如说通信双方没有任何一方的 Socket 固定，就好比打电话的双方彼此都不知道对方的电话号码，要通话是不可能的。

### 17.1.2　客户端 / 服务器模型

通常所说的 Socket 模式指的是利用 Socket 进行连接的模式，即客户端——服务器模型，简称 C / S 模型。C / S 模型主要有两种，一种是面向连接的 C / S 模型，另外一种是非面向连接模型。

下面我们分别介绍这两种模型：

1. 面向连接的 TCP 流方式

图 17-1 给出了采用面向连接协议时使用 Socket 的详细流程，读者从中可以知道使用 TCP

方式时，客户端和服务器端程序的全部处理过程。

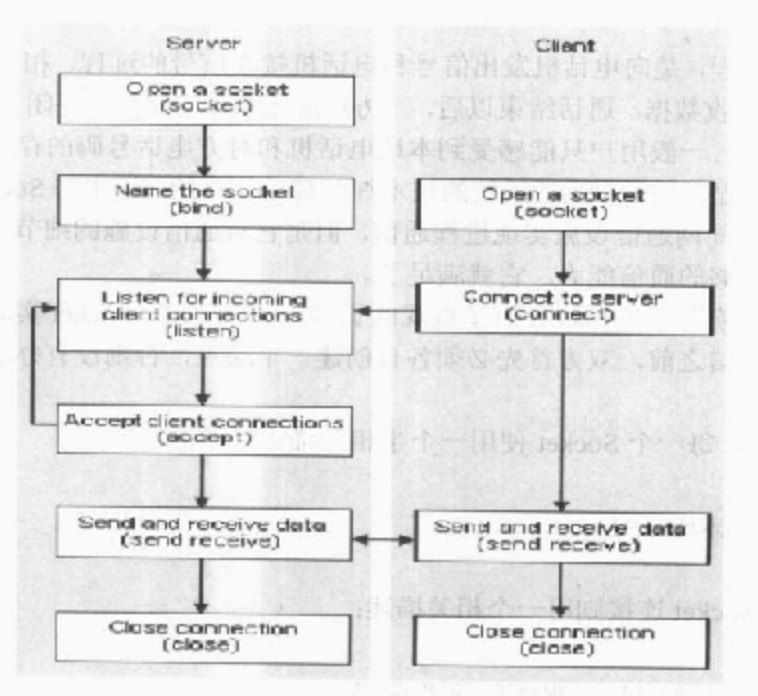



图 17-1　采用面向连接协议时 Socket 的使用

在这种模型中，客户端和服务器端的程序流程分别是：

（1）服务器程序流程

step 1: 程序初始化

step 2: 填写本机地址信息；

step 3: 绑定并在一个固定端口上进行监听；

step 4: 收到 client 的连接以后建立一个新的 Socket 连接；

step 5: 产生一个新的进程与 client 进行通信的信息处理；

step 6: 子通信程序结束后中断与 client 的连接。

（2）客户端程序流程

step 1: 程序初始化；

step 2: 填写服务器地址信息；

step 3: 连接服务器；

step 4: 与服务器通信和信息处理；

step 5: 通信结束后断开连接。

2. 无连接的 UDP（用户数据报）模式

图 17-2 给出了采用无连接协议时使用 Socket 的详细流程，读者从中可以知道使用 UDP

模式时，客户端和服务器端程序的全部处理过程。

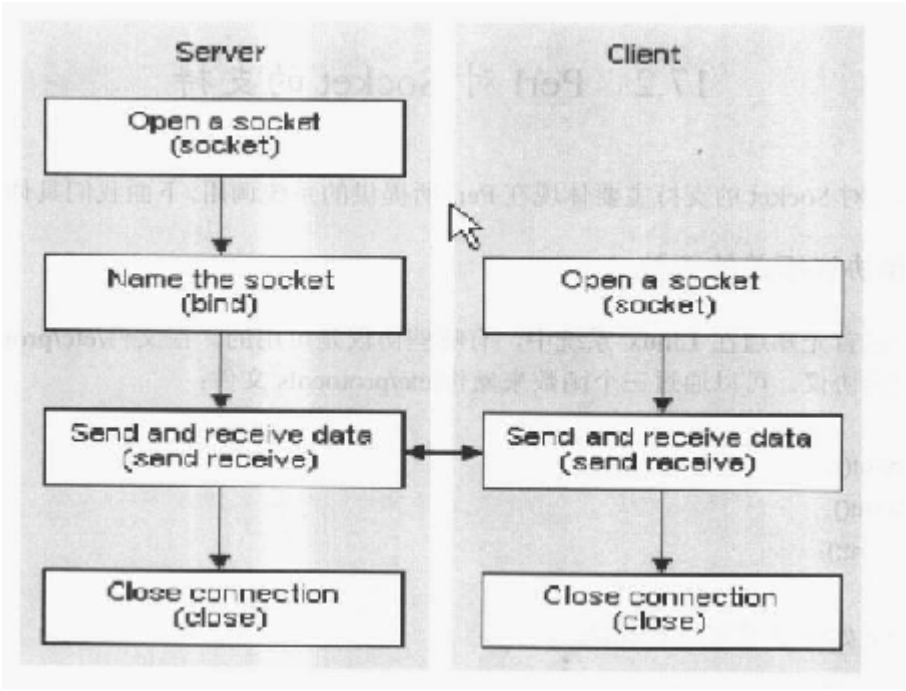



图 17-2　采用无连接协议的 Socket 的使用

在这种模型中，客户端和服务器端的程序流程分别是：

（1）服务器程序流程（单进程）

step 1: 程序初始化 ；

step 2: 填写本机地址信息 ；

step 3: 绑定一个固定的端口 ；

step 4: 收到 Client 的数据报后进行处理与通信 ；

step 5: 通信结束后断开连接。

（2）客户端程序流程：

step 1: 程序初始化 ；

step 2: 填写服务器地址信息 ；

step 3: 连接服务器 ；

step 4: 与服务器通信和信息处理 ；

step 5: 通信结束后断开连接 。

其实我们可以看到，UDP 方式下服务器与客户端程序差别不大，仅仅第三步不同而已。

## 17.2 Perl 对 Socket 的支持

Perl 语言对 Socket 的支持主要体现在 Perl 所提供的系统调用，下面我们具体的来看一看。

### 17.2.1 和协议相关的函数

读者应该首先知道在 Linux 系统中，有哪些协议是可用的。在文件/etc/protocols 里包含了这些可用的协议。可以通过三个函数来操作/etc/protocols 文件：

```
setprotoent();
getprotoent();
endprotoent();
```

请看一下如何使用这三个函数：

**程序清单 17-1：proto.pl**

```
#!/usr/bin/Perl
#
# List all the protocols in this machine
#
setprotoent(1);
while (($name, $aliases, $protonum) = getprotoent) {
        print " Name=$name, Aliases=$aliases, Protocol=$protonum \n";
}
endprotoent;
```

这个程序显示了可用的协议，它的执行结果如下：

```
Name=ip, Aliases=IP, Protocol=0
Name=icmp, Aliases=ICMP, Protocol=1
Name=igmp, Aliases=IGMP, Protocol=2
Name=ggp, Aliases=GGP, Protocol=3
Name=tcp, Aliases=TCP, Protocol=6
Name=pup, Aliases=PUP, Protocol=12
```

```
Name=udp, Aliases=UDP, Protocol=17
Name=idp, Aliases=IDP, Protocol=22
Name=raw, Aliases=RAW, Protocol=255
```

另外，也可以知道指定的协议是否可用。这可以通过 getprotobyname（）或者是 getprotobynumber（）函数来实现。

使用范例如下：

```
if  (($name, $aliases, $protonum) = getprotobyname('tcp')) {
        print "\n Name=$name, Aliases=$aliases, Protocol=$protonum";
}
print "\n";
```



getprotobyname()函数的参数是不区分大小写的，所以你也可以用'TCP'作为它的参数。

有许多函数的调用可以让你知道在你客户机上提供了哪些服务。这些函数查询一个名叫 /etc/services 的配置文件。程序清单 17-2 的程序演示了使用这几个函数的方法，先说明一下：

setservent()　函数让你把指针索引（一个抽象的索引）重新定位到文件的某一行，具体为哪一行由参数来指定。和这个函数一起的还有另外两个函数，分别是：

```
getservent()
endservent()
```

程序如 17-2 所示：

**程序清单 17-2：serve.pl**

```
#!/usr/bin/Perl
setservent(1);
printf "%15s %15s %4s %15s\n", "Name","Aliases","Port","Protocol";
while(($nm,$al,$pt,$pr) = getservent) {
        # print "Name=$nm, Aliases=$al, Port=$pt, $Protocol=$pr\n";
     printf "%15s %15s %4d %15s\n", $nm,$al,$pt,$pr;
}
endservent;
#end of this program
```

执行这个程序，读者将会看到如下的结果：

| Name | Aliases | Port | ;Protocol | |
|---|---|---|---|---|
| tcpmux | | 1 | tcp | |
| echo | | 7 | tcp | |
| echo | | 7 | udp | |
| discard | sink null | 9 | tcp | |
| discard | sink null | 9 | udp | |
| systat | users | 11 | tcp | |
| daytime | | 13 ; | tcp | |
| daytime | | 13 ; | udp | |
| netstat | | 15 ; | tcp | |
| qotd | quote | 17 | tcp | |
| msp | | 18 | tcp | |
| msp | | | 18 | udp |
| chargen | ttytst source | 19 | tcp | |
| chargen | ttytst source | 19 | udp | |
| ftp | | 21 | tcp | |
| telnet | | 23 | tcp | |
| smtp | mail | 25 &nbs p; | tcp | |
| time | timserver | 37 | tcp | |
| time | timserver | 37 | udp | |
| rlp | resource | 39 | udp | |
| nameserver | name | 42 &nbs p; | tcp | |
| whois | nicname | 43 | tcp | |
| domain | nameserver | 53 &n bsp; | tcp | |
| domain | nameserver | 53 &n bsp; | udp | |
| mtp | | 57 | tcp | |
| bootps | | 67 | tcp | |
| bootps | | 67 | udp | |
| bootpc | | 68 | tcp | |
| bootpc | | 68 | udp | |
| tftp | | 69&n bsp; | udp | |
| gopher | | 70 | tcp | |
| gopher | | 70 | udp | |
| rje | netrjs | 77 | tcp | |
| finger | | 79 | tcp | |
| www | http | 80 | tcp | |

| | | | | |
|---|---|---|---|---|
| www | | | 80 | udp |
| link | ttylink | 87 | & nbsp; | tcp |
| kerberos | krb5 | | 88 | tcp |
| kerberos | | 88 | &nbs p; | udp |
| supdup | | | 95 | tcp |
| hostnames | hostname | | 101 | tcp |
| iso-tsap | tsap | | 102 | tcp |
| csnet-ns | cso-ns | | 105 | tcp |
| csnet-ns | cso-ns | | 105 | udp |
| rtelnet | | 107 | &nbs p; | tcp |
| rtelnet | | 107 | &nbs p; | udp |
| pop2 | postoffice | | 109 | tcp |
| pop2 | | 109 | & nbsp; | udp |
| pop3 | | 110 | & nbsp; | tcp |
| pop3 | | 110 | & nbsp; | udp |
| sunrpc | | | 111 ; | tcp |
| sunrpc | | | 111 ; | udp |
| auth tap ident | authentication | | 113 | tcp |
| sftp | | 115 | & nbsp; | tcp |
| uucp-path | | | 117 | tcp |
| nntp readnews untp | | | 119 | tcp |
| ntp | | | 123 | tcp |
| ntp | | | 123 | udp |
| netbios-ns | | | 137 & nbsp; | tcp |
| netbios-ns | | | 137 & nbsp; | udp |
| netbios-dgm | | | 138 | tcp |
| netbios-dgm | | | 138 | udp |
| netbios-ssn | | | 139 | tcp |
| netbios-ssn | | | 139 | udp |
| imap2 | | | 143 | tcp |
| imap2 | | | 143 | udp |
| snmp | | | 161 & nbsp; | udp |
| snmp-trap | snmptrap | | 162 | udp |
| cmip-man | | | 163 &nb sp; | tcp |
| cmip-man | | | 163 &nb sp; | udp |
| cmip-agent | | | 164 & nbsp; | tcp |
| cmip-agent | | | 164 & nbsp; | udp |
| xdmcp | | | 177 | tcp |

| | | | |
|---|---|---|---|
| xdmcp | | 177 | udp |
| nextstep | NeXTStep NextStep | 178 | tcp |
| nextstep | NeXTStep NextStep | 178 | udp |
| bgp | | 179 | tcp |
| bgp | | 179 | udp |
| prospero | | 191 &nb sp; | tcp |
| prospero | | 191 &nb sp; | udp |
| irc | | 194 | tcp |
| irc | | 194 | udp |
| smux | | 199 & nbsp; | tcp |
| smux | | 199 & nbsp; | udp |
| at-rtmp | | 201 &nb sp; | tcp |
| at-rtmp | | 201 &nb sp; | udp |
| at-nbp | | 202 &nbs p; | tcp |
| at-nbp | | 202 &nbs p; | udp |
| at-echo | | 204 &nb sp; | tcp |
| at-echo | | 204 &nb sp; | udp |
| at-zis | | 206 &nbs p; | tcp |
| at-zis | | 206 &nbs p; | udp |
| z3950 | wais | 210 | tcp |
| z3950 | wais | 210 | udp |
| ipx | | 213 | tcp |
| ipx | | 213 | udp |
| imap3 | | 220 | tcp |
| imap3 | | 220 | udp |
| ulistserv | | 372 | tcp |
| ulistserv | | 372 | udp |
| exec | | 512 | tcp |
| biff | comsat | 512 | udp |
| login | | 513 | tcp |
| who | whod | 513 | udp |
| shell | cmd | 514 | tcp |
| syslog | | 514 &nbs p; | udp |
| printer | spooler | 515 | tcp |
| talk | | 517 | udp |
| ntalk | | 518 | udp |
| route | router routed | 520 | udp |
| timed | timeserver | 525 | udp |

| | | | |
|---|---|---|---|
| tempo | newdate | 526 | tcp |
| courier | rpc | 530 | tcp |
| conference | chat | 531 | tcp |
| netnews | readnews | 532 | tcp |
| netwall | | 533 &nb sp; | udp |
| uucp | uucpd | 540 | tcp |
| remotefs | rfs_server rfs | 556 | tcp |
| klogin | | 543 &nbs p; | tcp |
| kshell | | 544 &nbs p; | tcp |
| kerberos-adm | | 749 &nbs p; | tcp |
| Webster | | 765 &nb sp; | tcp |
| Webster | | 765 &nb sp; | udp |
| ingreslock | | 1524 ; | tcp |
| ingreslock | | 1524 ; | udp |
| prospero-np | | 1525 | tcp |
| prospero-np | | 1525 | udp |
| rfe | | 5002 | tcp |
| rfe | | 5002 | udp |
| krbupdate | kreg | 760 | tcp |
| kpasswd | kpwd | 761 | tcp |
| eklogin | | 2105 &n bsp; | tcp |
| supfilesrv | | 871 | tcp |
| supfiledbg | | 1127 ; | tcp |



上面列出来的就是作者的机器所能提供的服务。为什么有些公司内部的计算机可以上网（使用 HTTP 服务）但是却不可以连接 FTP 呢，原因就是它没有提供 FTP 这个服务，或者说，这项服务被关闭了。

除了上面讲到的函数以外，下面再来介绍两个函数：

gethostbyname() 参数为所要查询的主机的名字；

gethostbyaddr() 这个函数有两个参数，一个为所要查询的地址，另外一个为 AF_INET 的值。

下面看看这两个函数如何使用，程序如下：

```
#!/usr/bin/Perl
 $addr = pack('C4',204,251,103,2);
 ($name,$alias,$atype,$len,@addrs) = gethostbyaddr($addr,2);
 ($a,$b,$c,$d) = unpack('C4',$addrs[0]);
```

```
print "Name :   $name \n";
print "Alias:   $alias \n";
print "Type :   $atype \n";
print "Len  :   $len \n";
print "Node :   $a.$b.$c.$d \n";
$name = "www.tsinghua.edu.cn";
($name,$alias,$atype,$len,@addrs) = gethostbyname($name);
($a,$b,$c,$d) = unpack('C4',$addrs[0]);
print "Name :   $name \n";
print "Alias:   $alias \n";
print "Type :   $atype \n";
print "Len  :   $len \n";
print "Node :   $a.$b.$c.$d \n";
```

### 17.2.2 Socket 底层函数

1. Socket()

Socket()系统调用为服务器端和客户端程序生成一个套接口，定义如下：

```
Socket(SOCKET_HANDLE, $FAMILY, $TYPE, $PROTOCOL);
```

这个函数没有返回值，如果出错，可以查看$!变量，以确定是哪里出的错。Socket() 函数可以下面这种方式调用：

```
Socket(MY_HANDLE,  $AF_VNIX, $STREAMS, $PROTOCOL) ||
    die "\nCannot open Socket: $!\n";
print "\nSocket successfully opened\n";
```


你必须用 MY_HANDLE 描述符指向创建的 Socket，因为你将在程序的其他地方如子程序中用到它，注意这个 Socket 没有名字。

2. Bind()

bind() 系统调用给一个未命名的由 Socket() 创建的 Socket 一个名字，定义如下：

```
bind(SOCKET_HANDLE, $nameAsAString);
bind(SOCKET_HANDLE, $sockaddr);
```

第一个参数是由 Socket() 创建的句柄，如果读者使用 AF_VNIX，那么对应的第二个参数 $nameAsAString 指的就是要给这个 Socket 的名字。如果读者使用 AF_INET，指的就是它的地址。



AF_VNIX，AF_INET 中的 AF 指的是地址家族（Address Family），INET 指的是用于 Internet 协议的地址格式。

Bind() 系统调用方式如下：

- 对应于 AF_VNIX；
- bind(MY_HANDLE,"./mySocket") || die "Cannot bind $!\n"；
- 对应于 AF_INET。

```
$port = 6666
$AF_INET=2;       # Use AF_INET instead of AF_VNIX
($name,$alias,$atype,$len,$addr)=gethostbyname(`uname -n`);
$my_ip_addr = pack(S n C4 x8,$AF_INET,$STREAMS,$port,$addr);
bind(MY_HANDLE,$my_ip_addr) || die "Cannot bind $!\n";
```

这里要解释一下 pack()函数。此函数将数组或者列表的值，按照定义结构的模板格式压缩成二进制结构的格式，pack()函数调用方式如下：

```
pack(“指定的格式”，LIST);
```

使用范例如下：

```
$string = pack("C",65);       # 此时$string='A';
```

这条语句的意思是，将 65 这个 ASCII 码转换成一个 unsigned 字符，其中“C”参数就是指定要转换成 unsigned 字符的意思。

在这个例子中，我们期望的是一个 sockaddr 结构，如果用 C 语言表示就是：

```
{
    unsigned short family;
    int network;
    long address;
    char nullbytes[8];
}
```

3. Listen()、accept()

注意，这两个系统调用是用于服务器端程序的。

Listen() 函数被服务器程序用来侦听连接请求的，定义如下：

```
listen(SOCKET_HANDLE,$queueSize);
```

SOCKET_HANDLE 参数是 Socket 描述符，$queueSize 是等待连接的队列的最大长度。Listen() 函数典型调用如下：

```
listen(MY_HANDLE,5)

or die "Cann't listen $!\n";
```

如果 listen() 函数调用成功，就可以调用 accept() 函数接收连接了，如下所示：

```
accept(NEWSOCKET,ORIGINAL_SOCKET);
```

4. Connect()

Connect() 函数用于在面向连接的模型中，客户端需要向服务器端连接的时候调用，具体的调用方法如下：

```
Connect(SOCKET_HANDLE,"pathname");      # for AF_VNIX
Connect(SOCKET_HANDLE,$Address);        # for AF_INET
```

5. Getpeername()

该函数的调用返回 SOCKET 连接的另一端压缩形式的 Socket 地址，并且可以从中获取对方的名字，函数参数为 Socket 句柄。调用方法如下：

```
getpeername(SOCKET);
```

使用范例如下：

```
use Socket;
$peersockaddr = getpeername(SOCK);
($port, $ipaddr) = unpack_sockaddr_in($peersockaddr);
$peerhostname = gethostbyaddr($ipaddr,AF_INET);
```

```
$peeraddr = inet_ntoa($ipaddr);
```

6. Getsockname()

返回已经打开的 SOCKET 句柄在本地一端的压缩地址。调用方法如下：

```
getsockname(SOCKET);
```

使用范例如下：

```
use Socket;
$mysockaddr = getsockname(MYSOKCET);
($port, $myaddr) = unpack_sockaddr_in($mysockaddr);
```

7. Getsockopt()

返回有关 SOCKET 选项的信息，如果调用失败，则返回未定义值 undef，调用方法如下：

```
getsockopt(SOCKET,LEVEL,OPTNAMWE)
```

使用范例如下：

```
use Socket;
$mysockopt = getsockopt(SOCK,$NUM,$OPTNAME);
```

8. Recv()

Recv()函数调用方式如下：

```
recv(SOCKET,SCALAR,LEN,FLAGS)
```

此函数从第一个参数所指定的 SOCKET 句柄中读取指定字节的信息，并保存到指定的缓冲区中，利用 FLAYS 参数实现对数据的“偷看”功能。也就是说，在读取数据之后，并不会从系统缓冲区中删除这些已经读走的信息。此函数调用成功返回信息发送者的地址，失败则返回 undef。

使用范例如下：

```
use Socket;
while( $count &&select($rout = $rin,undef,undef,10.0) ) {
    $rtime = '';
    ($peeraddr = recv(SOCKET,$rtime,4,0))    or die "recv:$!";
      ($port,$peeraddr) = sockaddr_in($peeraddr);
      $host = gethostbyaddr($peeraddr,AF_INET);
      $peertime = unpack("N",$rtime) - $SECS_of_70_YEARS;
      printf " %-12s",$host;
      printf"%8d %s\n",$peertime –time,scalar localtime($peertime);
      $count--;
}
```

9. send()

send()函数的调用方法如下：

```
OCKET,MSG,FLAGS,TO)
```

send()在连接好的 Socket 句柄上发送数据。SOCKET 参数指定用户数据发送的 Socket 句柄；MSG 参数是包含数据的标量缓冲区；FLAGS 是可选参数，用于发生测试性的数据；TO 也是可选参数，向未建立连接的 Socket 发送数据。

使用范例如下：

```
foreach $host (@ARGV) {
  $count+;
  $peeriaddr = inet_aton($host) or die "unknown host";
  $peerpaddr = sockaddr_in($port, $peeriaddr)
  defined(send(SOCK,0,0,$peerpaddr)) or die"send $host:$!";
}
```

10. setsockopt()

调用方式如下：

```
setsockopt(SOCKET,LEVEL,OPTNAME,OPTVAL)
```

此函数设置 Socket 选项，调用失败，则返回 undef。OPTVAL 参数用于设置选项的数值，在有些情况下可能不需要，此时可以将其指定为 undef。

11. shutdown

shutdown()函数的调用方式如下：

```
shutdown(SOCKET,HOW)
```

调用此函数可以关闭所有或者是全部的全双工 Socket 连接。HOW 参数指定以什么样的方式关闭 Socket 连接：

0 —— 不再接收任何数据。
1 —— 不再发送任何数据。
2 —— 不再接收和发送任何数据。
如：

```
shutdown(MYSOCKET,1)  # 将不可以在这个套接口上接收数据
```

12. Socketpair()

建立一对连接的 Socket。调用方法如下：

```
Socketpair(SOCK1,SOCK2,DOMAIN,TYPE,PROTOCOL)
```

使用范例如下：

```
use Socket;
Socketpair(Rdr,Wtr,Aff_VNIX,SOCK_STREAM,PF_UNSPEC);
shutdown(Rdr,2);
shutdown(Wtr,0);
```

## 17.3　面向连接模型应用

通过前面的基础知识的介绍，这一节用 Perl 来写一个基于面向连接模型的应用。程序分为两个部分，一个是服务器端的程序，另外一个是客户端的程序。

### 17.3.1 服务器端的程序

程序代码如程序清单 17-3 所示：

**程序清单 17-3：server.pl**

```
#!/usr/bin/Perl
 #
 # Sample connection oriented server using Perl
 #
 $AF_VNIX = 1;
 $AF_INET=2;        # Use AF_INET instead of AF_VNIX.
 $SOCK_STR = 1;   # Use STREAMS.
 $PROTOCOL = 0;   # stick to the default protocols (IP).

 $port = 6668 unless $port;

 #
 # The pattern for packing into a sockaddr structure
 #
 $PACKIT='S n C4 x8';

 #
 # Disable any buffering on any newly created Sockets.
 #
 select(NEWSOCKET);
 $| = 1;
 select(stdout);

 #
 # Create the Socket.
 #
 Socket(MY_SOCKET, $AF_INET, $SOCK_STR, $PROTOCOL) ||
                              die "\n $0: Cannot open Socket: $!";
 print "Socket successfully opened\n";

 #
 # Get the host address for this node
 #
```

```
($name, $aliases, $addrtype, $len, @addrs) = gethostbyname("www.ikra.com");
($a,$b,$c,$d) = unpack('C4',$addrs[0]);
print "Server Name=$name, Server Address= $a.$b.$c.$d\n";
$my_ip_addr = pack($PACKIT,$AF_INET,$port,$addrs[0]);

#
# If you just want to test with the localhost, try this line
# instead of the above.
# $my_ip_addr = pack($PACKIT,$AF_INET,$port,127,0,0,1);

#
# Bind to the Socket and listen on this port
#
bind(MY_SOCKET, $my_ip_addr) || die "$0: Cannot bind .. $!\n";
print   "\n Bound to Socket";
listen(MY_SOCKET,5)   || die "$0: Cannot listen: $!\n";
print   "\n Listening \n";

while(1) {
      $remote = accept(NEWSOCKET, MY_SOCKET) || die "$0: Unacceptable: $!\n";
         # In case you have to display incoming connection
         # information, you can uncomment the next three lines of code:
         # @remoteInfo = unpack($PACKIT,$remote);
      # $, = ' ';
      # print @remoteInfo; print "\n";
      # $pid = fork || &cleanup;
      if ($pid == fork)   {   # child
            sleep 3;
            print NEWSOCKET "Welcome to this server\n";
            # in child,.. you can do other stuff here.
            close NEWSOCKET;
            # I chose to just print this message and terminate
            close MY_SOCKET;
            exit;
      }
      else {   # parent s
            sleep 10;
```

```
            close MY_SOCKET;
            close NEWSOCKET;   # in parent
            exit;
        }
    }
sub cleanup {
        close NEWSOCKET; close(MY_SOCKET); die "$0: Cannot fork : $!\n";
    }
```

可以看到，在上面的程序中，服务器通过以下调用来建立连接：

```
Socket()
bind()
listen()
accept()
```

当服务器端和客户端的连接建立成功以后，双方可以提供 read() 和 write()系统调用来交换数据。

从 Socket 上读取数据，可以调用 read()函数实现：

```
read(MY_SOCKET,$buffer,$length);
```

往 Socket 上写数据，可以调用 write()实现

```
write(MY_SOCKET,$buffer,$length);
```

当服务完成以后，必须关闭 Socket，以释放系统资源：

```
close(MY_SOCKET);
```

此外，还可以使用 shutdown()系统调用，来限制某一个端口上的数据接收和发送，如下所示：

```
shutsown(MY_SOCKET,HOW);
```

如果 HOW 为 0，限制数据接收；HOW 为 1，限制数据发送；HOW 为 2，同时限制数据的接收和发送。

### 17.3.2 · 客户端的程序

客户端程序代码如下：

**程序清单 17-4：client.pl**

```
#!/usr/bin/Perl
 #
 # define constants for talking to server
 #
 $PACKIT = 'S n C4 x8';
 $AF_INET = 2;
 $SOCK_STR = 1;        # STREAMS
 $DEF_PROTO = 0;       #default protocol of IP
 $port = 6668 unless $port;

 #
 # Get the name of the server
 #
 ($name, $aliases, $addrtype, $len, @addrs) = gethostbyname("www.ikra.com");
 ($a,$b,$c,$d) = unpack('C4',$addrs[0]);
 print "Server Name=$name, Server Address= $a.$b.$c.$d\n";
 $that = pack($PACKIT,$AF_INET,$port,$addrs[0]);

# Confine yourself to the local host machine address of 127.0.0.1
   $this = pack($PACKIT,$AF_INET,$port,127,0,0,1);

# Disable buffering on this Socket.
   select(CLIENT_SOCKET);
 $| = 1;
 select(stdout);

 Socket(CLIENT_SOCKET,$AF_INET,$SOCK_STR,$DEF_PROTO)
or   die "$0: Cannot open Socket\n";
 print "Created Socket\n";
```

```
 # Attempt to connect to server.
do {
sleep(1);
  $result = connect(CLIENT_SOCKET,$that);
  if (result != 1) {
       print "Sleeping\n";
  }
} while ($result != 1);

sleep(5);
print "After connection\n";

# send data to server
# write(CLIENT_SOCKET,"hello",5);
read(CLIENT_SOCKET,$buf,100);
print "[" . $buf . "]\n";
close(CLIENT_SOCKET);
# end of client program
```

上面程序通过以下步骤建立连接并收发数据：

```
Socket()
pack() ;   # 获得一个结构，以标志自身和将要连接的服务器
connect();
read() 或者 write();
```



要养成良好的习惯，当使用完一项服务以后，一定要释放曾经申请的系统资源，以便其他的程序可以使用。在这里其实只需要简单的执行一个 Close(SOCKET)系统调用就可以了。

## 17.4 使用预编译模块

许多客户端软件，如邮件客户端软件（foxmail,outlook）、Web 浏览器、新闻组阅读器等都是用 TCP/IP 协议和 Socket 编程接口实现的。CPAN（Complete Perl Archive Network）上有几个功能库提供了这些客户端的库函数，这样读者在编写自己的客户端软件时，可以不必关心应用层的协议。

### 17.4.1　使用 Net::FTP 模块

此模块实现了客户端的文件传输协议，使用范例如下：

```
use Net::FTP;
$ftp = Net::FTP->new(www.ftpsite.com);
die "Could not connect :$! "  unless  $ftp;
$ftp->login("anonymous","youname@263.net");
$ftp->cwd('/pub/movie/');     # 使用cwd()函数改变工作目录
$ftp->get('main.html');
$ftp->quit();
```

此模块支持所有可以从标准的 FTP 程序输入的命令，如 put、get、ls 等等。你可以使用此模块来编写自己的 ftp 客户端软件，是不是很有成就感！

### 17.4.2　使用 Net::POP3 模块

通过此模块提供的函数，可以从 POP(Post Office Protcol)服务器上读取邮件，使用范例如程序清单 17-5 所示：

**程序清单** 17-5：pop3.pl

```
use Net::POP3;
$mail = Net::POP3->new($POP3Server');
die "could not open account" unless $mail;
$n = $mail->login($account,$passwd);
$r_msgs = $mail->list();  # 返回一个指向散列表的引用
foreach $msg_id (keys %$r_msgs)
{
  print "Msy $msg_id (",$r_msgs->{$msg_id},"):\n";
  $rl_msg = $mail->top($msg_id,3);
  print @$rl_msg;
}
$mail->quit;
```



想要了解此模块的更多信息，请访问 CPAN 站点，并获取相关的文档。

# 17.5 RPC 的实现

这一节，重点介绍一个 RPC（远程过程调用）模块，它的实现是基于 Perl 对 Sockets 的支持。

RPC 模块的实现分两部分来完成。首先实现一个异步的消息传递系统，即 Msg 模块，它的实现采用了非阻塞的 I/O；然后通过继承 Msg 模块来实现 RPC（remote procedure call）。这个例子综合使用了前面所讲的基础知识，如匿名存储、子程序引用等 Perl 高级应用。仔细研究这个例子，一定会受益匪浅。

## 17.5.1 Msg 模块实现

Msg 模块对外提供了如下一些接口函数：

（1）connect(host,port,[rcv_callback])函数

此函数连接到地址为 host、而端口为 port 的服务器并返回一个连接对象。rcv_callback()函数为指向用户定义的子程序的引用，当对端进程发送一个消息时，它将以如下形式被调用：rcv_callback($conn,$msg,$err)，其中 conn 为连接对象，用来发送消息和断开连接；msg 为接收到的消息，如果对方关闭了连接，则 msg 内容为 undef，如果有 rcv_callback 参数的话，$err 包含最近一次 sysread 的错误值。

（2）$conn->send_now($msg)函数

立即把队列中的消息发送出去。如果队列中存在已经在排队的消息，那么它会先把那些消息发送出去然后再发送$msg。

（3）$conn->send_later($msg)函数

这是一种非阻塞的发送消息的方式，这个函数把消息放在与连接对象相关联的队列中，并让 event_loop()函数在套接字可写时进行分发。因此必须在某个时刻调用 event_loop 函数，否则，消息将永远不会被发送出去。

（4）$conn->disconnect()函数

断开连接。

（5）($msg,$err) = $conn->rev_now()函数

以阻塞的方式接收消息直到接收完一个完整的消息为止。在标量变量的上下文中，此函数返回消息，否则返回任何存在的错误代码。

（6）new_server($thishost,$thisprot,[login_proc]) 函数

此函数在地址为 thishost 的 thisport 端口上创建一个监听套接口。当远端的套接字企图连接时，将以连接对象和连接主机及端口号为参数调用 login_proc 函数，如果 login_proc 返回 undef，则连接被关闭。

（7）set_event_handler($handle,["read"=>rd_cb],["write=>wt_cb]) 函数

handle 参数是一个句柄对象，如套接字、文件或者是管道句柄，也可以是从 IO::Handle 导出的对象。当相应的文件描述符准备好读写时，event_loop 将会调用回调函数，当回调值为 undef 时，已注册的回调将被删除。对于一个给定的句柄只能注册一个特定类型的回调函数。

（8）event_loop([count]) 函数

循环执行 count 次所给定的代码。当为事件循环注册的句柄不存在时，循环退出。

Msg 模块的实现分为四个部分：

- 发送端子程序：用于连接一个远程过程，并向它发送消息。
- 接收端子程序：用于当消息和连接请求到来时通知一个已注册的子程序。
- 对非阻塞 I/O 支持子程序：根据系统对 POSIX 模块的支持与否，建立一个阻塞或非阻塞的 Socket。
- 事件循环支持子程序：用于文件相关事件的发送。

首先来看看发送端的子程序是如何实现的：

```
package Msg;
use strict;
use IO::Select;
use IO::Socket;
use Carp;
use vars qw(%rd_callbacks %wt_callbacks $rd_handles $wt_handles);
%rd_callbacks = ();
%wt_callbacks = ();
$rd_handles    = IO::Select->new();
$wt_handles    = IO::Select->new();
my $blocking_supported = 0;
```

上面这些代码表明 Msg 是以模块的形式实现的，具体的接口函数的实现，将在后面详细介绍。

### 1. Msg 发送端子程序实现

首先是 connect 函数。

**程序清单 17-6：connect.pl**

```
sub connect {
    # @_特殊变量用来接收传递给connect函数的参数，其中$_[0]表示第一个参数，#以此类推
    my ($pkg, $to_host, $to_port,$rcvd_notification_proc) = @_;
```

```
    # 创建一个新的Internet域的Socket套接字
    my $sock = IO::Socket::INET->new (
                                        PeerAddr => $to_host,
                                        PeerPort => $to_port,
                                        Proto    => 'tcp');
    return undef unless $sock;
    # 创建一个连接对象
    my $conn = bless {
        'sock'                      => $sock,
        'rcvd_notification_proc' => $rcvd_notification_proc,
    }, $pkg;
      if ($rcvd_notification_proc) {
        # 将_rcv和$conn一起绑定在一个closure中
        my $callback = sub {_rcv($conn)};
        set_event_handler ($sock, "read" => $callback);
    }
    $conn;  # 表明connect函数返回的是$conn连接对象
}
```

connect 子程序建立一个客户端的套接口，并且创建一个连接对象。这个连接对象是一个通信的端点，它具有如下属性：

sock —— Socket 对象。

rcvd_notification_proc —— 当接收到消息时调用的函数。

queue —— 指向缓冲消息数组的引用。

send_offset —— 在非阻塞模式下，如果套接字发生了阻塞，就记下队列最顶层的消息发送了多少。

msg —— 在非阻塞模式下，msg 中包含了部分到达的消息。

bytes_to_read —— 包含了消息队列中对应于某一个消息还需要读取的字节数。

一旦连接建立，双方可以通过本地的连接对象进行通信。当用户指定了回调函数 $rcvd_notification_proc，便建立了一个事件处理程序来调用私有函数_rcv，它会在一个完整的消息被接收到以后接着调用这个回调函数。这在 connect 函数的 if 条件语句里实现。

这里解释一下 bless。bless 关键字在内部将对象标记为指向它们所在包的指针。bless 以一个普通的指向数据结构的引用为参数，它会把那个数据结构（注意，不是引用本身）标记为属于某个特定的包。bless 的好处在于它提供给我们一种直接使用该对象的方式。

下面是 disconnect()函数和 send_now()函数的实现代码：

**程序清单 17-7：disconnect.pl**

```
sub disconnect {
```

```
        my $conn = shift;
    my $sock = delete $conn->{sock};
    return unless defined($sock);
        set_event_handler ($sock, "read" => undef, "write" => undef);
    close($sock); undef $!; # Should ideally process errors from close
}
# send_now()函数实现
sub send_now {
    my ($conn, $msg) = @_;
    _enqueue ($conn, $msg);
    $conn->_send (1);   # 1 ==> flush
}
```

send_now()函数将消息放入队列，并告诉_send 函数去刷新此消息以及已经在队列中排队的消息。

```
sub send_later {
    my ($conn, $msg) = @_;
    _enqueue($conn, $msg);
    my $sock = $conn->{sock};
    return unless defined($sock);
    set_event_handler ($sock, "write" => sub {$conn->_send(0)});
}
```

send_later()函数将消息放入队列，并注册一个执行“写”的回调函数。这个回调函数将在以后调用 event_loop()且文件描述符可写时被调用。

```
sub _enqueue {
    my ($conn, $msg) = @_;
    # prepend length (encoded as network long)
    my $len = length($msg);
    $msg = pack ('N', $len) . $msg;
    push (@{$conn->{queue}}, $msg);
}
```

_enqueue()函数在每个消息前面加上四个字节的标示此消息长度的信息，然后把整个消息放到队列中去。使用了 pack 函数对长度信息进行编码，在接收端用 unpack 函数进行解码。注意，消息本身并没有进行编码，因为消息本身是不用考虑字节顺序的。

程序清单 17-8：_send.pl

```
sub _send {
    my ($conn, $flush) = @_;
    my $sock = $conn->{sock}; # conn为连接对象，这是散列表的引用语句
    return unless defined($sock);
    my ($rq) = $conn->{queue};    #指向消息队列的引用
#   如果设置了$flush，则将套接字设置为阻塞模式，于是将所有队列中的消息发送出去，只有
#   在发生错误时返回；如果将 $flush设置为0，那么套接字将被设置为非阻塞模式，这样只有
#   在每个消息发送后或者是有可能在发送消息当中阻塞时，才返回事件循环
    $flush ? $conn->set_blocking() : $conn->set_non_blocking();
    my $offset = (exists $conn->{send_offset}) ? $conn->{send_offset} : 0;
    while (@$rq) {
        my $msg           = $rq->[0];
        my $bytes_to_write = length($msg) - $offset;
        my $bytes_written  = 0;
        while ($bytes_to_write) {
            $bytes_written=syswrite($sock,$msg, $bytes_to_write, $offset);
            if (!defined($bytes_written)) {
                if (_err_will_block($!)) {
                    #   应当只发生在延迟模式，记下已经发送的数量
                    $conn->{send_offset} = $offset;
                    # 由于事件处理程序已经进行了设置，因此最终要进行回调并恢复发送
                    return 1;
                } else {      # Uh, oh
                    $conn->handle_send_err($!);
                    return 0; # fail. Message remains in queue ..
                }
            }
            $offset += $bytes_written;
            $bytes_to_write -= $bytes_written;
        }
        delete $conn->{send_offset};
        $offset = 0;
        shift @$rq;
        last unless $flush; # Go back to select and wait for it to fire again.
    }
    #   如果队列中的消息没有发送完，就回调次函数
```

```
    if (@$rq) {
        set_event_handler ($sock, "write" => sub {$conn->_send(0)});
    } else {
        set_event_handler ($sock, "write" => undef);
    }
    1;  # 类似于return 1;语句，表示此函数执行成功
}
```

_send 函数完成实际的消息发送工作，它由 send_now()函数调用或者是作为事件循环的回调函数被调用。如果是从 send_now 函数中调用，它会将套接字设置为阻塞模式并刷新队列中的所有消息；如果是从事件循环中被调用，它会将套接字设置为非阻塞模式，并在返回事件循环以前每次最多刷新队列中的一条消息。如果 sysread()函数表示要阻塞了，_send 会记下这条消息已经发送了多少并返回事件循环。

```
sub handle_send_err {
    my ($conn, $err_msg) = @_;
    warn "Error while sending: $err_msg \n";
    set_event_handler ($conn->{sock}, "write" => undef);
}
```

这个错误处理函数只是关掉事件通知，没有做别的错误处理。

2. Msg 接收端子程序实现

下面这些子程序是实现了接收端的功能，主要是用来侦听接收。

```
my ($g_login_proc, $g_pkg);          # 前缀g表示global，这是代码风格的问题
my $main_Socket = 0;
# 首先是new_server函数实现
sub new_server {
    @_ == 4 || die "new_server (myhost, myport, login_proc)\n";
    my ($pkg, $my_host, $my_port, $login_proc) = @_;

    $main_Socket = IO::Socket::INET->new (
                                        LocalAddr => $my_host,
                                        LocalPort => $my_port,
                                        Listen    => 5,
                                        Proto     => 'tcp',
                                        Reuse     => 1);
```

```
    die "Could not create Socket: $! \n" unless $main_Socket;
    set_event_handler ($main_Socket, "read" => \&_new_client);
    $g_login_proc = $login_proc; $g_pkg = $pkg;
}
```

new_server 函数类似于 connect 函数，它用创建一个监听套接字并用事件句柄注册一个用户定义的登录过程。当服务器端程序调用 event_loop 函数并且一个连接请求到来的时候，这个登录函数就会被调用。但是，new_server 函数并不创建一个连接对象，创建连接对象是由_new_client 函数实现的，如程序清单 17-9 所示：

**程序清单 17-9：new_client.pl**

```
sub _new_client {
    my $sock = $main_Socket->accept();
    my $conn = bless {
        'sock' =>   $sock,
        'state' => 'connected'
    }, $g_pkg;
      my $rcvd_notification_proc = &$g_login_proc ($conn);
    if ($rcvd_notification_proc) {
        $conn->{rcvd_notification_proc} = $rcvd_notification_proc;
        my $callback = sub {_rcv($conn)};
        set_event_handler ($sock, "read" => $callback);
    } else {   # Login failed
        $conn->disconnect();
    }
}
```

当收到一个连接请求时，_new_client 函数被调用。当执行完 accept 以后，用户定义的登录函数将有机会接收或者是拒绝一个连接。这个代码引用立即与新建的连接对象相关联，并在消息到达那个连接时被调用。_rcv 函数注册为标准的回调函数，由它来处理所有到来的消息，并在积蓄了一条完整的消息之后才调用上面所提到的代码引用。

程序清单 17-10 是_rcv 函数的实现：

**程序清单 17-10：disconnect.pl**

```
sub _rcv {
    my ($conn, $rcv_now) = @_;
    my ($msg, $offset, $bytes_to_read, $bytes_read);
    my $sock = $conn->{sock};
```


http://www.yurennet.com

```
return unless defined($sock);
if (exists $conn->{msg}) {
    $msg          = $conn->{msg};
    delete $conn->{'msg'};
    $offset       = length($msg) - 1;
    $bytes_to_read = $conn->{bytes_to_read};
} else {
    # 这是典型的情况
    $msg          = "";
    $offset       = 0 ;
    $bytes_to_read = 0 ;                    # Will get set soon
}
  #以阻塞模式来读取消息的长度信息，这里假设在读取四个字节信息的时间内时不会阻塞的
if (!$bytes_to_read)  {   # 获取新的长度信息
    my $buf;
    $conn->set_blocking();
    $bytes_read = sysread($sock, $buf, 4);
    if ($! || ($bytes_read != 4)) {
        goto FINISH;
    }
    $bytes_to_read = unpack ('N', $buf);
}
$conn->set_non_blocking() unless $rcv_now;
while ($bytes_to_read) {
    $bytes_read = sysread ($sock, $msg, $bytes_to_read, $offset);
    if (defined ($bytes_read)) {
        if ($bytes_read == 0) {
            last;
        }
        $bytes_to_read -= $bytes_read;
        $offset        += $bytes_read;
    } else {
        if (_err_will_block($!)) {
            # 只能以非阻塞模式运行到这里
            $conn->{msg}          = $msg;
            $conn->{bytes_to_read} = $bytes_to_read;
            return ;# 返回事件循环，当套接字再次可以读时，_rcv将会被调用
        } else {
```

```
                    last;
                }
            }
        }
        # 消息成功读取
    FINISH:
        if (length($msg) == 0) {
            $conn->disconnect();
        }
        if ($rcv_now) {
            return ($msg, $!);
        } else {
            &{$conn->{rcvd_notification_proc}}($conn, $msg, $!);
        }
    }
```

_read 和 _send 函数是一个函数对，一个是发送字节流，而另外一个是接收字节流。和 _send 函数不一样，_read 函数不知道有多少数据需要处理，但是它知道字节流中前面 4 个字节包含了编码过的消息的长度。为了简化处理，_read 函数首先设置为阻塞模式，然后读出字节流中前面 4 个字节的数据。和_send 函数一样，有可能读到的数据比实际请求的数据要少，也可能返回一个指示阻塞的错误。如果套接口阻塞，_rcv 函数复制缓冲区中的部分消息到连接对象，指出还有多少数据需要继续读出，然后返回，并且等待事件循环的激发再次读数据。如果出现错误，它自动取消连接。

下面是 rcv_now 函数实现：

```
sub rcv_now {
    my ($conn) = @_;
    my ($msg, $err) = _rcv ($conn, 1); # 1 means receive immediately
        return wantarray ? ($msg, $err) : $msg;
}
```

wantarry 是用来查明调用者指定的是上下文是数组变量还是标量变量。

3. Msg：对非阻塞 I／O 的支持

```
BEGIN {
    eval {
        require POSIX; POSIX->import(qw(F_SETFL O_NONBLOCK EAGAIN));
    };
    $blocking_supported = 1 unless $@;
}
```

BEGIN 子程序首先测试是否可以装载 POSIX 模块。如果可以装载的话，就设置 $blocking_supported 变量，这个变量则有下面几个子程序：

```
sub _err_will_block {
    if ($blocking_supported) {
        return ($_[0] == EAGAIN());
    }
    return 0;
}
sub set_non_blocking {
    if ($blocking_supported) {
        # 保留其他的fcntl标志位
        my $flags = fcntl ($_[0], F_GETFL(), 0);
        fcntl ($_[0], F_SETFL(), $flags | O_NONBLOCK());
    }
}
sub set_blocking {
    if ($blocking_supported) {
        my $flags = fcntl ($_[0], F_GETFL(), 0);
        $flags  &= ~O_NONBLOCK(); # 清除阻塞标志，但保留其余的标志
        fcntl ($_[0], F_SETFL(), $flags);
    }
}
```

可以看到，set_blocking 函数和 set_non_blocking 是通过调用系统函数 fcntl 来实现的。F_SETFL 参数把文件描述符的标志设置为位掩码（bitmask），所以必须小心操作，以免把设置号的文件描述符的标志更改了。

4. Msg：事件循环子程序的实现

事件循环子程序的实现使用了 IO::Select 模块管理文件句柄和套接口（Socket）句柄。它由两个子程序 set_event_handler 和 set_event_loop 完成。程序清单 17-11 是 set_event_handler() 函数的实现例子：

**程序清单 17-11**：set_event_handler.pl

```
sub set_event_handler {
        shift unless ref($_[0]); # 如果第一个参数为包名，就执行shift函数
    my ($handle, %args) = @_;
    my $callback;
        if (exists $args{'write'}) {
            $callback = $args{'write'};
```

```
            if ($callback) {
                $wt_callbacks{$handle} = $callback;
                $wt_handles->add($handle);
            } else {
                delete $wt_callbacks{$handle};
                $wt_handles->remove($handle);
            }
        }
        if (exists $args{'read'}) {
            $callback = $args{'read'};
            if ($callback) {
                $rd_callbacks{$handle} = $callback;
                $rd_handles->add($handle);
            } else {
                delete $rd_callbacks{$handle};
                $rd_handles->remove($handle);
            }
        }
    }
```

set_event_handler 函数通过使用句柄作为散列表的索引来记录 read 和 write 函数调用，可以使用回调值为 undef 作为参数来删除一个回调。

**程序清单 17-12：event_loop.pl**

```
sub event_loop {
        my ($pkg, $loop_count) = @_;
        my ($conn, $r, $w, $rset, $wset);
        while (1) {
            # Quit the loop if no handles left to process
            last unless ($rd_handles->count() || $wt_handles->count());
            ($rset,$wset)=IO::Select->select($rd_handles,$wt_handles, undef, undef);
            foreach $r (@$rset) {
                &{$rd_callbacks{$r}} ($r) if exists $rd_callbacks{$r};
            }
            foreach $w (@$wset) {
                &{$wt_callbacks{$w}}($w) if exists $wt_callbacks{$w};
            }
            if (defined($loop_count)) {
```

```
                last unless --$loop_count;
            }
        }
    }
```

这个函数表示无限循环，但是可以通过指令设置位有限次数地进行循环。这种指定循环次数的想法可以分发（dispatch）其他的事件而不会失去对无限循环的控制。

### 17.5.2　Msg 模块使用举例

这里举一个简单的使用 Msg 模块的例子，Msg 模块的使用分为客户端和服务器端两部分。

1. 使用 Msg 模块的客户端

```
use Msg;
$conn = Msg->connect('localhost', 8080);
die "Error: Could not connect\n" unless $conn;
$conn->send_now("Message $i");
($msg, $err) = $conn->rcv_now();
```

connect()函数创建一个连接对象，send_now()函数把消息发送到那个连接上，而一个相应的名为 rcv_now()的函数将阻塞直到另一方收到一个消息为止。

2. 使用 Msg 模块的服务器端

```
use Msg;
use strict;
my $host = 'localhost';
my $port = 8080;
Msg->new_server($host, $port, \&login_proc);
print "Server created. Waiting for events";
Msg->event_loop();
#-------------------------------------------------------------
sub login_proc {
    # 无条件的接收到来的连接请求
    return \&rcvd_msg_from_client;
}
```

```
sub rcvd_msg_from_client {
    my ($conn, $msg, $err) = @_;
    if (defined $msg) {
        print "$msg\n";
    }
}
```

上面程序调用 new_server()函数创建一个监听套接字，并调用了 event_loop 函数进行循环处理。

### 17.5.3 RPC 模块的实现

本小节将要用到上一小节设计的 Msg 模块以及 Perl 的 FreezeThaw 模块来实现一个远程过程调用模块——RPC.pm。

远程过程调用的含义是指可以透明的调用另外一个进程空间中的函数，就像调用本进程空间的函数一样。由于使用了 Msg 模块和 FreezeThaw 模块，这样，RPC 模块的实现变得很简单了。

1. 实现 RPC 模块的调用端接口

**程序清单 17-13：call_rpc.pl**

```
package RPC;
use Msg;
use strict;
use Carp;
@RPC::ISA = qw(Msg);
use FreezeThaw qw(freeze thaw);
sub connect {
    my ($pkg, $host, $port) = @_;
    my $conn = $pkg->SUPER::connect($host,$port, \&_incoming_msg);
    return $conn;
}
my $g_msg_id = 0;
my $send_err = 0;
sub handle_send_err {
    $send_err = $!;
}
sub rpc {
```

```
    my $conn = shift;
    my $subname = shift;

    $subname = (caller() . '::' . $subname) unless $subname =~ /:/;
    my $gimme = wantarray ?  'a' : 's';  # 数组或标量变量
    my $msg_id = ++$g_msg_id;
    my $serialized_msg = freeze ('>', $msg_id, $gimme, @_);
    # 发送与接收
    $conn->send_later ($serialized_msg);
    do {
        Msg->event_loop(1); # Dispatch other messages until we
                            # get a response
    } until (exists $conn->{rcvd}->{$msg_id} || $send_err);
    if ($send_err) {
        die "RPC Error: $send_err";
    }
    # 消息出队
    my $rl_retargs = delete $conn->{rcvd}->{$msg_id}; # 指向数组的引用
    if (ref($rl_retargs->[0]) eq 'RPC::Error') {
        die ${$rl_retargs->[0]};
    }
    wantarray ? @$rl_retargs : $rl_retargs->[0];
}
```

connect()函数只是简单的调用 Msg 模块的 connect 函数，并将_incoming_msg 作为所有消息到来时被调用的子程序。Handle_send_err 函数重载了 Msg::handle_send_err，并将遇到的任何错误保存起来。而该错误代码在 rpc 函数中进行检查。在 rpc 函数中使用了 FreezeThaw 模块的 freeze 函数将如下一些信息捆绑到一个大的字符串中。这些信息包括：远程子程序的名称、子程序的参数、wantarray 指示、请求“>”或应答“<”指示以及消息标志。

在 rpc 函数中，使用了 Msg->send_later()，这是因为它尽可能地触发非阻塞 I/O，只有当 event_loop 被调用时，消息才真正被发送出去，因为要由它来判断套接字什么时候可写。与此同时，event_loop 用来跟踪其他到来的消息并把它们分发出去。计数器为 1 将迫使事件循环在分发完一个回合的消息后立即返回，这样程序便可以获得控制权。当连接对方机器的响应到来时，event_loop 将调用_incoming_msgs 函数，由它对响应进行解码，并将返回值和连接对象关联起来。下面是为服务器端调用提供的接口函数的实现。

2. 实现 RPC 模块的服务器端接口

**程序清单 17-14：** server_rpc.pl

```
sub new_server {
    my ($pkg, $my_host, $my_port) = @_;
    $pkg->SUPER::new_server($my_host, $my_port,
                            sub {$pkg->_login(@_)});
}
sub _login {
    \&_incoming_msg;
}
sub _incoming_msg {
    my ($conn, $msg, $err) = @_;
    return if ($err);    # Need better error handling.
    return unless defined($msg);
    my ($dir, $id, @args) = thaw ($msg);
    my ($result, @results);
    if ($dir eq '>') {
        # New request message
        my $gimme = shift @args;
        my $sub_name = shift @args;
        eval {
            no strict 'refs';  # 因为使用符号引用来调用这个子程序
                               # 希望返回一个数组
            if ($gimme eq 'a') {  # Want an array back
                @results = &{$sub_name} (@args);
            } else {
                $result = &{$sub_name} (@args);
            }
        };
        if ($@) {
            $msg = bless \$@, "RPC::Error";
            $msg = freeze('<', $id, $msg);
        } elsif ($gimme eq 'a') {
            $msg = freeze('<', $id, @results);
        } else {
```

```
            $msg = freeze('<', $id, $result);
        }
        $conn->send_later($msg);
    } else {
        # Response to a message we had sent out earlier
        $conn->{rcvd}->{$id} = \@args;
    }
}
```

上面代码实现了在默认情况下，所有的接入连接都被无条件的接收，并且将消息传递给子程序_incoming_msg 处理，在 new_server()函数中通过$pkg 间接地调用_login 函数。

_incoming_msg 函数对应于 rpc 方法，它打开由 rpc 发送的消息并检查其发送方向（是请求还是应答）。如果是请求，它就通过符号引用来调用所需要的子程序，根据 wantarry 的指示，提供了标量变量或数组变量形式的结果；如果 eval 代码块报告了错误，那么变量$@将被填上 RPC::Error 的错误标志并返回给调用进程。

### 17.5.4　RPC 模块使用举例

RPC 模块的使用同样包括客户端的应用和服务器端的应用。

1. 客户端应用

```
use RPC;
my $conn = RPC->connect($host, $port);
my $answer = $conn->rpc('ask_sheep',
                        "Ba ba black sheep, have you any wool ?");
print "$answer\n";
```

connect 函数根据给定的主机和端口号，建立一个 RPC 连接，然后通过 rpc 函数进行远程过程的调用。

2. 服务器端应用

对应于上面的客户端应用，服务器端提供所需的子程序，并调用 new_server 和 event_loop 函数来接收到来的 RPC 调用，这样 ask_sheep 函数会被很好地调用。服务器端代码如下：

```
RPC->new_rpc_server($host, $port);
RPC->event_loop();

sub ask_sheep {  # Sample subroutine to be invoked from client
    print "Question: @_\n";
    return "No";
```

```
}
```

## 17.6 小结与练习

### 17.6.1 小结

本章讲解了如何利用 Perl 对 Socket 的支持进行网络编程，透彻地解释了 Socket 的基本原理以及建立在此基础上的面向连接模型；然后在此基础上设计了面向连接模型的应用，以及 RPC 模块的实现。其中 RPC 模块的实现使用了许多 Perl 的高级应用，如匿名存储，子程序引用等。此外还通过具体的例子介绍了如何利用预编译模块进行客户端程序的编写。

### 17.6.2 习题与思考

（1）试着用 HTTP 协议编写一个 Web 客户程序。
（2）使用 IO::Socket 模块，编写一个扫描服务器端口的程序。

# 附录

## 部分习题参考答案



# 第 1 章　习题参考答案

（1）参考方案如下：

首先要建立一个硬件环境。考虑到 Linux 系统对机器硬件的要求不是很高，可以用一台旧的或者是淘汰的机器做服务器（当然，如果条件优越的话就无所谓了），在这个机器上安装 Linux 系统（推荐使用 RedHat Linux,但是 rh 版本的 Linux 不支持中文，所以必须自己安装中文字库，这对于初学者有点难度，所以，也可以使用 TurboLinux 或者是红旗 Linux）和 Perl 编译器；然后用另外一台机器安装 Windows9x、NT 或者是 W2K，作为客户端，两台机器的连接可以使用串口或者是通过网卡使用交叉线相连（对于使用水晶头的网线）。如果 Linux 服务器安装了 Samba 服务器，那么通过配置它的配置文件可以从 Windows 系统上直接访问 Linux 系统上的文件，这样，你可以使用 UltraEdit 之类的软件进行程序编辑，当然也可以使用 telnet 命令或者是 NetTerm 软件登录到 Linux 系统，使用 Vi 编辑器进行程序编辑。此外，如果使用串口相连的话，还可以省去一台显示器，只需要主机就可以。

为了有效的学习 perl 编程，你可能需要更多的资料，所以建议阅读许多其它的与 Linux 及 Perl 文化、技术相关的书籍和访问这一类的站点，这样读者可以有效的进行学习。

# 第 2 章　习题参考答案

（1）答案为：

```
2.33e3   表示的十进制数为：   2330
-0.673e-2   表示的十进制数为：-0.00673
0377   表示的十进制数为：  255
-0xff   表示的十进制数为：  -255
```

（2）答案为：

```
'hello'      # 五个字符: h, e, l, l, o
'don\'t'     # 五个字符: d, o, n,单引号和 t
''           # 空
'silly\\me'  #  silly,反斜杠和me
"\nhow are you\n"  # 换行符，how are you, 再加一个换行符
"all\tright"    # all, 一个制表符,和 right
```

```
"d:\\perl"        # d:, 反斜杠, 和perl
```

（3）答案为：

```
$a = "fred";    # $a现在是 "fred"
$b = "some text $a";        # $b 现在是 "some text fred"
$c = "no such variable $what";    # $c 现在是 "no such variable "
$x = '$fred';   # 一个$符号和fred
$y = "hey $x"; # $y现在是hey $fred
$fred = 'hi';     # $fred 现在是hi
$barney = "a test of \$fred";     # $barney现在是 a test of $fred
```

（4）参考答案为：

```
$radius = 7.6;
print "The radius is: $radius \n";
$pi = 3.141592654;
$result = 2 * $pi * $radius;
print "The circumference with the radius $radius is $result\n";
```

# 第3章 习题参考答案

（1）屏幕上打印的结果为：

```
Bright OrangeBurntCanary YellowSunbeamBright OrangeBurnt Middle Canary YellowSun
beam
12789101415
ABFGHYZ
```

（2）在屏幕上打印的结果为：

```
The length of the array is : 3
one two three
1324
```

（3）在屏幕上打印的结果为：

```
couch chair table stove
couch chair table stove
chair couch table stove
```

（4）参考答案为：

```
print "Enter the list of strings:\n";
@list = <STDIN>;
@reverselist = reverse @list;
print @reverselist;
```

程序执行过程及输出如下：

```
Enter the list of strings:
How are you?
Fine,and you?
I am fine,too.

I am fine,too.
Fine,and you?
How are you?
```

（5）参考答案为：

```
srand;
print "List of strings: "; @b = <STDIN>;
print "Answer: $b[rand(@b)]";
```

# 第 4 章　习题参考答案

（1）打印结果为：

```
Joe's birthday is: June 2
Jennifer's birthday is: Mar 20
```

（2）一种参考答案为：

```
%map = qw(red apple green leaves blue ocean, yellow ,Banana, orange, Carrot);
print "A string please: ";
chomp($some_string = <STDIN>);
print "The value for $some_string is $map{$some_string}\n";
```

（3）在屏幕上输出结果为：

```
Known foods:
Banana
```

```
Apple
Raspberry
Carrot
Lemon
```

（4）参考答案为：

```
chomp(@words = <STDIN>); # 读入单词，删掉换行符
foreach $word (@words) {
  $count{$word} = $count{$word} + 1; # 或 $count{$word}++
}
foreach $word (keys %count) {
  print "$word was seen $count{$word} times\n";
}
```

该程序执行结果如下：

```
iek       # 输入单词，每个单词输入完后按回车输入下一个
i
a
idke
b
a
i
a
b
c
idke      # 按结束输入符
iek was seen 1 times      # 输出统计信息
a was seen 3 times
i was seen 2 times
b was seen 2 times
c was seen 1 times
idke was seen 2 times
```

# 第 5 章　习题参考答案

（1）屏幕上打印的结果为：

```
17  20
```

```
3  7
7  7
5  5
4  4
100
a1
Ba
aaa
```

（2）参考答案为：

```
print "First number: ";
chomp($a = <STDIN>);
print "Second number: ";
chomp($b = <STDIN>);
$c = $a * $b;
print "Answer is $c\n";
```

（3）参考答案为：

```
print "String: ";
$a = <STDIN>;
print "Number of times: ";
chomp($b = <STDIN>);
$c = $a x $b; print "The result is:\n$c";
```

（4）屏幕上打印的结果为：

```
X20
fredfredfred
barneybarneybarney
```

6. 程序输出为：

```
579
123456
1284
321321321321
24
2.41785163922926e+024
118
20
```

# 第6章 习题参考答案

（1）参考答案为：

```
print "What temperature is it? ";
chomp($temperature = <STDIN>);
if ($temperature > 30) {
    print "Too hot!\n";
} elsif ($temperature < 10) {
    print "Too cold!\n";
} else {
    print "Just right!\n";
}
```

（2）参考答案为：

```
print "Enter a number (999 to quit): ";
chomp($n = <STDIN>);
while ($n != 999) {
  $sum += $n;
  print "Enter another number (999 to quit): ";
  chomp($n = <STDIN>);
}
print "the sum is $sum\n";
```

程序执行过程如下：

```
Enter a number (999 to quit): 1
Enter another number (999 to quit): 3
Enter another number (999 to quit): 99
Enter another number (999 to quit): -8
Enter another number (999 to quit): 23
Enter another number (999 to quit): 999
the sum is 118
```

（3）程序输出为：

```
0 1 2 3 4 5 5 6 7 8 9 10
0 1 2 3 4 ** ** 6 7 8 9 10
```

（4）参考答案为：

```
print "Enter some strings, end with ^Z:\n";
@strings = <STDIN>;
while (@strings) {
  print pop @strings;
}
```

程序执行过程如下：

```
Enter some strings, end with ^Z:
abc
ok
sorry
thank you
yes
yes
thank you
sorry
ok
abc
```

（5）输出结果为：

```
0123456789
5678910
11  12  14  16  18  20  12  14  16  18
5678910
```

（6）参考答案为：

a. 不使用列表的程序：

```
for ($number = 0; $number <= 20; $number++) {
  $square = $number * $number;
  printf "%5g %8g\n", $number, $square;
}
```

b. 使用列表的程序：

```
foreach $number (0..20) {
  $square = $number * $number;
  printf "%5g %8g\n", $number, $square;
}
```

程序执行结果如下：

| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 4 |
| 3 | 9 |
| 4 | 16 |
| 5 | 25 |
| 6 | 36 |
| 7 | 49 |
| 8 | 64 |
| 9 | 81 |
| 10 | 100 |
| 11 | 121 |
| 12 | 144 |
| 13 | 169 |
| 14 | 196 |
| 15 | 225 |
| 16 | 256 |
| 17 | 289 |
| 18 | 324 |
| 19 | 361 |
| 20 | 400 |

# 第7章 习题参考答案

（1）程序输出的结果为：

```
Please input your operating system :
windows
Now you operating system is: Linux
```

（2）参考答案为：

```
(a)/a+b*/
(b)/\\*\**/
(c)/($whatever){3}/
(d)/[\000-\377]{5}/ 或者 /(.|\n){5}
```

（3）这里给出三种参考答案：

方法一：

```
while (<STDIN>) {
    if (/a/i && /e/i && /i/i && /o/i && /u/i) {
        print;
    }
}
```

方法二：

```
while (<STDIN>) {
    if (/a.*e.*i.*o.*u/i) {
        print;
    }
}
```

方法三：

```
while (<STDIN>) {
    if (/^[eiou]*a[^iou]*e[^aou]*i[^aeu]*o[^aei]*u[^aeio]*$/i) {
        print;
    }
}
```

（4）参考答案为：

```
/com(poundl)/;        # 匹配 "compound" 或 "com"
/com(pound(sl)l)/;  # 匹配"compounds", "compound" 或 "com"
/com(pound)?/;        #  匹配"compound" 或 "com"
/com(pound(s?))?/;  # 匹配"compounds", "compound" 或 "com"
/com(pounds?)?/;     # 同上
```

（5）程序输出结果为：

```
I am 24 high and 80 long
This Is A Long Line
```

（6）程序输出结果为：

```
Found number 1752
Found number 10
Found number 3
Found rd after the last number.
```

# 第 8 章 习题参考答案

（1）程序输出为：

Dromedary
Dromedary
DROMEDARY
DROMEDARY
dROMEDARY

（2）各个语句的作用效果如下：

```
tr/aeiou/!/;                    # 将元音字母变成!
tr{/\\\r\n\b\f. }{_};           # 将上述特殊字符变成下划线
tr/A-Z/a-z/ ;          # 将大写字母变成小写字母
$count = tr/0-9//;              # 统计变量$_中的数字
tr/@$%*//d;                     # 删除字符@$%*
```

（3）程序输出结果为：

Found love at 2 in I love you!.
Found love at 2 in I love you!.

（4）程序输出结果为：

sure
sure
The string before replace :I am OK!
After replaced: I am O
The return is K
Before replace: Thank you!
After replaced : Thank

# 第 9 章　习题参考答案

（1）参考答案为：

```
$pi = atan2(1,1) * 4;
$piover180 = $pi/180;
sub acos {
        atan2( sqrt(1 - $_[0] * $_[0]), $_[0] )
   }
for ($deg = 0; $deg <= 45; $deg++) {
     printf "%3d %7.5f\n", $deg, cos($deg * $piover180);
     printf "%3d %7.5f\n", $deg, acos($deg * $piover180);
}
```

（2）参考答案为：

```
sub max {
     my $max = shift(@_);
     for my $item (@_) {
          $max = $item if $max < $item;
     }
     return $max;
}
@number = (-9,78,0,34,8,0.9e3,-4.567,2.3);
$max = max( @number);
print "The max of (";
foreach $num(@number) {
   my $i = 0;
   $i++;
   print $num," ";
}
print ") is $max.\n";
```

程序执行结果为：

```
The max of (-9 78 0 34 8 900 -4.567 2.3 ) is 900.
```

（3）参考答案为：

```
sub upcase_in {
    for (@_) { tr/a-z/A-Z/ }
}
$v1 = "Are you full of OK?";
$v2 = "All right!";
print "The original string is : \n";
print "$v1\n$v2\n";
upcase_in($v1, $v2);   # this changes $v1 and $v2
print "Now the string is : \n";
print "$v1\n$v2\n";
```

程序执行输出为：

```
The original string is :
Are you full of OK?
All right!
Now the string is :
ARE YOU FULL OF OK?
ALL RIGHT!
```

（4）参考答案为：

```
use Time::localtime;
($day, $month, $year) = (localtime)[3,4,5];
printf("The current date is %04d %02d %02d\n", $year+1900, $month+1, $day);
$tm = localtime;
print "Today is day ", $tm->yday, " of the current year.\n";
```

（5）输出结果为：

```
There were 265333775 seconds between Nat and Bree
(438 weeks, 4 days, 23:49:35)
```

（6）参考答案一：

```
sub hypotenuse {
    return sqrt( ($_[0] ** 2) + ($_[1] ** 2) );
}
$diag = hypotenuse(3,4);   # $diag is 5
```

参考答案二：

```
sub hypotenuse {
    my ($side1, $side2) = @_;
    return sqrt( ($side1 ** 2) + ($side2 ** 2) );
```

```
}
print hypotenuse(3, 4), "\n";              # prints 5
@a = (3, 4);
print hypotenuse(@a), "\n";                # prints 5
```

# 第 10 章　习题参考答案

（1）程序输出为：

```
fred  barney
george  jane  elroy
homer  marge  bart
```

（2）程序输出为：

```
123 Anylane Rd.
123 Allways Dr.
```

（3）程序输出为：

```
SCALAR ARRAY HASH
CODE REF
```

# 第 11 章　习题参考答案

（1）参考答案为：

```
print "What file? ";
chomp($filename = <STDIN>);
open(THATFILE, "$filename") ||
  die "cannot open $filename: $!";
while (<THATFILE>) {
  print "$filename: $_";
}
```

（2）参考答案为：

```
print "Input file name: ";
chomp($infilename = <STDIN>);
```

```
print "Output file name: ";
chomp($outfilename = <STDIN>);
print "Search string: ";
chomp($search = <STDIN>);
print "Replacement string: ";
chomp($replace = <STDIN>);
open(IN,$infilename) || die "cannot open $infilename for reading: $!";
die "will not overwrite $outfilename" if -e $outfilename;
open(OUT,">$outfilename") || die "cannot create $outfilename: $!";
while (<IN>) { # read a line from file IN into $_
  s/$search/$replace/g; # change the lines
  print OUT $_; # print that line to file OUT
}
close(IN);
close(OUT);
```

（3）参考答案为：

```
while (<>) {
  chomp; # eliminate the newline
  print "$_ is readable\n" if -r;
  print "$_ is writable\n" if -w;
  print "$_ is executable\n" if -x;
  print "$_ does not exist\n" unless -e;
}
```

（4）参考答案为：

```
while (<>) {
  chomp;
  $age = -M;
  if ($oldest_age < $age) {
    $oldest_name = $_;
    $oldest_age = $age;
  }
}
print "The oldest file is $oldest_name ", "and is $oldest_age days old.\n";
```

# 第 12 章　习题参考答案

（1）参考答案为：

```
$file = shift || die "usage: $0 filename";
open(F, $file) || die "open: $!";
while (<F>) {
  ($user, $company, $email) = split /:/;
  write;
}
format STDOUT =
@<<<<<<<<<<<<<< @<<<<<<<<<<<<<<<<<<<<<<< @<<<<<<<<<<<<<<<<<
$user,          $company,                 $email
```

（2）参考答案为：

```
# append to program from the first problem...
format STDOUT_TOP =
User            Company                   Real Name
==============  ========================  =================
```

（3）参考答案为：

```
# append to program from the first problem...
format STDOUT_TOP =
Page @<<<
$%
User            Company                   Real Name
==============  ========================  =================
```

（4）参考答案为：

```
print reverse <>;
```

说明：这个程序虽然只有一行，但却能够完成所需的功能，原因是输入的各行都存在数组@ARGV 中，当输入结束后，就将数组@ARGV 颠倒输出。

程序执行情况如下：

```
ok
yes
sorry    # 输入结束
sorry    # 输出
yes
ok
```

# 第 13 章 习题参考答案

（1）参考答案如下：

在一般情况下，Perl 在对文件进行语法分析时会编译整个文件，而编译成功以后，从第一条全局语句往后开始执行。但是，如果在语法分析时碰到了名为 BEGIN 的子程序或代码块，它不仅将其编译，还会立刻执行，然后再接着对剩余文件的编译工作。所以可以使用 BEGIN 子程序来做一些初始化的工作。

同样的，如果要在程序的所有代码执行完毕时作一些收尾工作，可以使用 END 代码块。因为当程序要退出时，无论它是成功退出还是由于别的原因退出，END 代码块都会被调。

此外，Perl 还支持多条 BEGIN 和 END 语句。BEGIN 语句根据其出现的先后顺序来执行，而 END 语句则以出现的先后顺序的相反顺序来执行。

（2）参考答案如下：

package 是简单的命名空间管理设备，比如说，它可以让同一个程序中的两个部分同时有叫做$same 的不同变量。

函数库是许多特定用途函数的集合。函数库本身常常声明为独立的包，让相关的变量和函数放在一起。旧式的函数库会用独立的文件存放，文件名通常以“.pl”结尾；如果要在程序中用到其中的函数，必须通过 require 函数。

模块是符合某种特定写法的函数库，用户程序可以利用 use 关键字，在编译时利用其中的函数。模块名通常以“.pm”结尾，这是 use 的规定。

Pragma 是一种会在编译时和执行时对程序产生影响的模块，pragma 和其它模块不同，通常会把它的影响代码块局限在程序的一小部分。Pragma 的名字习惯上全部小写。

# 第 14 章　习题参考答案

（1）参考答案为：

定义类的代码如下：

```
package Person;
sub new {
    my $class = shift;
    my $self  = {};
    return bless $self, $class;
}
sub name {
    my $self = shift;
    $self->{NAME} = shift if @_;
    return $self->{NAME};
}
sub age {
    my $self = shift;
    $self->{AGE} = shift if @_;
    return $self->{AGE};
}
```

测试类的代码如下：

```
use Person;
my $dude = Person->new();
$dude->name("Jason");
$dude->age(23);
  printf "%s is age %d.\n", $dude->name, $dude->age;
```

（2）参考答案为：

Employee 类继承 Person 类的方法如下：

```
package Employee;
use Person;
@ISA = ("Person");
1;
```

测试代码如下：

```
use Employee;
my $empl = Employee->new();
$empl->name("Jason");
$empl->age(23);
printf "%s is age %d.\n", $empl->name, $empl->age;
```

# 第 15 章 习题参考答案

（1）参考答案如下：

```
use strict;
use CGI qw(:standard);
print header(), start_html("Add Me");
print h1("Add Me");
if(param()) {
        my $n1 = param('field1');
        my $n2 = param('field2');
        my $n3 = $n2 + $n1;
        print p("$n1 + $n2 = <strong>$n3</strong>\n");
} else {
        print hr(), start_form();
        print p("First Number:", textfield("field1"));
        print p("Second Number:", textfield("field2"));
        print p(submit("add"), reset("clear"));
        print end_form(), hr();
}
print end_html();
```

如果没有输入，程序会产生有两个输入字段的页面，如果有输入，程序就会打印出两个数相加的结果。

（2）参考答案如下（提示：使用 HTTP_USER_AGENT 环境变量）

```
use strict;
use CGI qw(:standard);
```

```
print header(), start_html("Browser Detective");
print h1("Browser Detective"), hr();
my $browser = $ENV{'HTTP_USER_AGENT'};
$_ = $browser;

BROWSER:{
    if (/msie/i) {
            msie($_);
    } elsif (/mozilla/i) {
            netscape($_);
    } elsif (/lynx/i) {
            lynx($_);
    } else {
            default($_);
    }
}

print end_html();
sub msie{
    print p("Internet Explorer: @_.  Good Choice\n");
}

sub netscape {
    print p("Netscape: @_.  Good Choice\n");
}

sub lynx {
    print p("Lynx: @_.  Shudder...");
}

sub default {
    print p("What the heck is a @_?");
}
```

设计这个程序关键在于 HTTP_USER_AGENT 环境变量。可能有些服务器软件没有实际运行这个变量，但是大部分都有。利用这个变量，可以为不同的浏览器设计不同的页面。

# 第 16 章 习题参考答案

（1）参考答案如下：

```
# 第一个程序
dbmopen(%WORDS,"words",0644);
while (<>) {
    foreach $word (split(/\W+/)) {
        $WORDS{$word}++;
    }
}
dbmclose(%WORDS);
```

第一个程序会在当前目录中产生名为 words.dir 和 words.pag 的 DBM 文件。然后 while 循环用<>运算读取每行数据，在用 split 函数以/\W+/为模式进行切割。这里用 DBM 哈希数组对所有 word 出现的次数计数。

```
#第二个程序
dbmopen(%WORDS,"words",undef);
foreach $word (sort { $WORDS{$b} <=> $WORDS{$a} } keys %WORDS) {
    print "$word $WORDS{$word}\n";
}
dbmclose(%WORDS);
```

第二个程序在当前目录下打开名为 words 的 DBM 文件，此程序的关键之处在于 foreach 循环。$word 值每次会是列表中下个元素的值，此列表是由%WORDS 的关键字根据其值大小排序而得。程序最后，打印出列表中每个字本身，以及它出现过的次数。

（2）参考答案如下：

```
dbmopen(%ALIAS, "/etc/aliases", undef) ||
    die "No aliases!: $!";
while (($key,$value) = each(%ALIAS)) {
    chop($key,$value);
    print "$key $value\n";
}
```

调用 dbmopen 函数打开 aliases 的 DBM 数据库，在 while 循环中，用 chop 出去 key 和 value 末端的 NUL 字符，在把结果打印出来。

# 第 17 章　习题参考答案

（1）参考答案如下：

要编写 Web 客户程序，只要与 Web 服务器建立连接，然后根据协议发出 HTTP 请求即可。可以使用底层的套接字来完成这个任务，但是 Perl 提供了现有的模块来做这些事情。下面给出一个示例程序：

```
use HTTP::Request;
use LWP::UserAgent;
$ua = new LWP::UserAgent();
$request = new HTTP::Request('Get','http://localhost/index.html');
$response = $us->request($request);
print $response->content;
```

这个程序从本地 Web 服务器中取回文件 index.html，并打印出来。$ua 是一个 LWP::UserAgent 对象，用于建立连接，想服务器发送请求；$request 是一个 HTTP::Request 对象，用于构造一个 HTTP 请求；$response 是一个 HTTP::Response 对象，用于分析 Web 服务器返回的数据。

（2）参考答案如下：

这里给出一个参考例程，名为 portscan.pl，如下所示：

```
#!/usr/bin/perl
use IO:Socket;
my ($line, $port, @servers);
# 接收命令行参数
($servers=$AGRV[0] || $usage;
$begin = ($ARGV[1] || 0);
# 扫描 1-1024 端口
for ($port = $begin; $port<=1024;$port++) {
    $sock = IO::Socket::INET->new(PeerAddr=>$server,
                                  PeerPort=>$port,
                                  Proto=>'tcp');
    if($sock) {
        print "端口 $port 可用 \n";
    }
```

```
}
sub usage{
    print <<EOF;
    用如下方式调用此程序：
    Perl portscan.pl www.somesite.com
    EOF
    exit 1;
}
```

此程序从命令行读入所要扫描地服务器地址，然后从端口 1 一直扫描到 1024，并打印出提供服务的端口号。

```
}
sub usage{
    print <<EOF;
    用如下方式调用此程序：
    Perl portscan.pl www.somesite.com
    EOF
    exit 1;
}
```

此程序从命令行读入所要扫描地服务器地址，然后从端口 1 一直扫描到 1024，并打印出提供服务的端口号。